# UNIVERSIDADE ABERTA


# BOOLEAN ALGEBRA: FROM DIGITAL CIRCUITS TO DEEP LEARNING APPLICATIONS

**Filipa Trindade Coito**

**Mestrado em Bioestatística e Biometria**

**Dissertation advised by Prof.ª Dr.ª Patrícia Engrácia and co-advised by Prof.ª Dr.ª Gilda Ferreira**

**November 2024**

# UNIVERSIDADE ABERTA

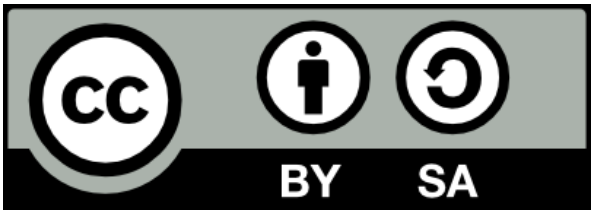

# BOOLEAN ALGEBRA: FROM DIGITAL CIRCUITS TO DEEP LEARNING APPLICATIONS

**Filipa Trindade Coito**

**Mestrado em Bioestatística e Biometria**

**November 2024**

## Creative Commons License



## Acknowledgments

I would like to express my sincere gratitude to everyone who contributed to the completion of this work and to my academic journey.

First and foremost, I thank my advisor and co-advisor, Prof. Drª. Patrícia Engrácia and Prof. Drª. Gilda Ferreira, respectively, for their support, guidance, and patience throughout each stage of this process. Their insights and expertise were invaluable to the development of this dissertation.

To my parents, who taught me resilience and shaped me into who I am today.

To my fiancé, for your unconditional love, emotional support, and constant encouragement, without which this journey would not have been possible. Thanks for always believing in me and motivating me to pursue my dreams, even when I doubt myself.

To my friends, who are there for me even after weeks of radio silence, my heartfelt gratitude. Thank you for your incredible support and friendship.

To you, Nuno.

Thank you.

UNIVERSIDADE ABERTA
www.uab.pt

**DECLARAÇÃO DE INTEGRIDADE**

**STATEMENT OF INTEGRITY**

Declaro ter atuado com integridade na elaboração da presente dissertação/tese. Confirmo que em todo o trabalho conducente à sua elaboração não recorri à prática de plágio ou a qualquer outra forma de falsificação de resultados.

Mais declaro que tomei conhecimento integral do Regulamento Disciplinar da Universidade Aberta, publicado no Diário da República, 2.ª série, n.º 215, de 6 de novembro de 2013.

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged Disciplinary Regulations of the Universidade Aberta (regulation published in the official journal Diário da República, 2.ª série, N.º 215, de 6 de novembro de 2013).

Universidade Aberta, 27 de Novembro de 2024

Nome completo/Full name: Filipa Trindade Coito

Assinatura/Signature:

Filipa Trindade

manuscrita ou digital / handwritten or digital

# Resumo

A álgebra de Boole tem um papel central na lógica digital, sendo essencial para o design e otimização de circuitos e sistemas computacionais. Este trabalho explora os princípios da álgebra de Boole e as suas aplicações tradicionais em circuitos digitais, bem como as suas utilizações inovadoras, como no campo do *deep learning*. Esta aplicação permite desenvolver modelos computacionalmente eficientes e avançar o estado da arte em áreas como sistemas embebidos e computação de baixa potência.

A dissertação analisa em profundidade a álgebra de Boole, examinando as suas propriedades e leis, incluindo a simplificação de expressões booleanas. São discutidos conceitos como a soma de produtos, o produto de somas e a construção de mapas de Karnaugh, fundamentais na otimização de circuitos digitais. Em seguida, aborda-se a aplicação da álgebra de Boole no design de circuitos digitais, com foco em circuitos combinacionais e sequenciais. Exemplos práticos, como somadores, codificadores e contadores binários, são apresentados para ilustrar como estas operações booleanas fundamentam a construção destes circuitos.

Por fim, o trabalho explora a relação entre a álgebra de Boole e as redes neuronais, com base no estudo de Petersen *et al.* (2023), que propõe redes neuronais baseadas em portas lógicas (LGNs). Estas redes são adaptadas para versões diferenciáveis, permitindo a utilização de métodos de otimização baseados em gradientes. O estudo analisa o desempenho das LGNs em problemas de classificação de imagens, destacando as suas vantagens e desafios em comparação com arquiteturas tradicionais.



**Palavras-chave:** *Álgebra de Boole, Redes Neuronais, Funções Lógicas, Deep Learning, Logic Gate Networks.*

# Abstract

Boolean algebra plays a central role in digital logic, being essential for the design and optimization of circuits and computational systems. This work explores the principles of Boolean algebra and its traditional applications in digital circuits, as well as its innovative uses, such as in the field of deep learning. These applications enable the development of computationally efficient models and advance the state of the art in areas like embedded systems and low-power computing.

The dissertation delves deeply into Boolean algebra, examining its properties and laws, including the simplification of Boolean expressions. Concepts such as sum of products, product of sums and the construction of Karnaugh maps are discussed, as they are fundamental in optimizing digital circuits. It then addresses the application of Boolean algebra in digital circuit design, focusing on combinational and sequential circuits. Practical examples, such as adders, encoders, and binary counters, are presented to illustrate how these Boolean operations underpin the construction of such circuits.

Finally, the work explores the relationship between Boolean algebra and neural networks, based on the study by Petersen *et al*. (2023), which proposes logic gate neural networks (LGNs). These networks are adapted to differentiable versions, enabling the use of gradient-based optimization methods. The study analyzes the performance of LGNs in image classification tasks, highlighting their advantages and challenges compared to traditional architectures.



**Keywords:** *Boolean Algebra, Neural Networks, Logic Functions, Deep Learning, Logic Gate Networks.*

# Resumo Alargado em Português

Os sistemas numéricos formam a base das operações matemáticas, oferecendo estruturas para representar e manipular quantidades. Do sistema decimal, que é o mais familiar, aos sistemas binário, octal e hexadecimal, cada um apresenta vantagens únicas para expressar e resolver problemas numéricos.

O sistema decimal, com base 10, está profundamente enraizado na vida quotidiana, estando na base de tarefas como contar, medir e fazer orçamentos. Sendo um sistema posicional, o valor de cada dígito depende da sua posição dentro de um número, tornando-o intuitivo e amplamente utilizado.

Em contraste, o sistema binário opera com uma base 2, utilizando apenas dois dígitos: 0 e 1. Este sistema é fundamental para a ciência da computação, alinhando-se perfeitamente com a natureza de dois estados dos dispositivos eletrónicos, como os transístores. A simplicidade do binário aumenta a fiabilidade na transmissão e processamento de dados, formando a base de toda a computação digital. Embora possa representar qualquer quantidade, este sistema exige mais dígitos que o sistema decimal para o efeito, mas a sua ligação com o design eletrónico torna-o indispensável. As conversões entre binário e decimal são habituais em informática, onde os humanos trabalham com números decimais, enquanto as máquinas processam números binários.

As operações binárias sustentam o funcionamento das portas lógicas nos circuitos digitais, modelados matematicamente através de operações lógicas.

A álgebra de Boole é uma estrutura matemática introduzida por George Boole em 1853, no seu trabalho *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities*. Este sistema revolucionou a lógica ao desenvolver uma álgebra para valores binários (verdadeiro e falso), representados por 1 e 0. As ideias pioneiras de Boole tornaram-se a base da lógica simbólica, mais tarde chamada de álgebra de Boole. Inicialmente teórica, esta álgebra encontrou aplicações práticas na década de 1930, quando Konrad Zuse desenhou a sua máquina de cálculo Z1 (Mano & Kime, 2014). Na mesma altura, pioneiros da computação digital como John Atanasoff, Alan Turing, Claude

Shannon e John von Neumann reconheceram o potencial da álgebra de Boole, utilizando-a para desenvolver circuitos lógicos e computadores com programas armazenados (Mano & Kime, 2014). Estes avanços estabeleceram a álgebra de Boole como um alicerce da computação digital.

No seu núcleo, a álgebra de Boole lida com variáveis binárias e operações lógicas. As três operações fundamentais são NOT, AND e OR. A operação NOT nega uma entrada: se $x = 1$, então $\bar{x} = 0$, e vice-versa. A operação AND gera 1 apenas quando ambas as entradas, $x$ e $y$, são 1. Finalmente, a operação OR gera 1 se pelo menos uma das entradas, $x$ e/ou $y$, for 1. Juntas, estas operações formam os blocos de construção para a criação de funções booleanas, que combinam variáveis e operações para representar expressões lógicas.

Para entender melhor estas operações, as tabelas da verdade oferecem uma representação clara das relações entre entradas e saídas. Por exemplo, a operação NOT inverte simplesmente o valor da entrada, enquanto as operações AND e OR funcionam de acordo com regras predefinidas.

A álgebra de Boole segue um conjunto de leis e propriedades que simplificam e manipulam expressões lógicas. Estas incluem a lei da identidade ($1x = x, 0 + x = x$), a lei do elemento nulo ($0x = 0, 1 + x = 1$), a lei do inverso ($x\bar{x} = 0, x + \bar{x} = 1$) entre outras, como as propriedades comutativa, associativa e distributiva. A Leis de De Morgan são particularmente significativas. Estas leis descrevem como a negação interage com as operações AND e OR:

- O complemento de uma operação AND é o OR dos complementos: $\overline{xy} = \bar{x} + \bar{y}$.
- O complemento de uma operação OR é o AND dos complementos: $\overline{x + y} = \bar{x}\bar{y}$.

Além dos circuitos lógicos, a álgebra de Boole tem fortes paralelos com a teoria dos conjuntos. Operações lógicas como NOT, AND e OR correspondem a operações de conjunto como complemento, interseção e união. Esta conexão permite aplicar os princípios booleanos em várias áreas da matemática, desde a resolução de problemas relacionados com conjuntos até à conceção de circuitos digitais.

A simplificação das expressões lógicas traz vantagens práticas, especialmente em computação. Expressões mais simples reduzem a complexidade dos circuitos, minimizam erros na implementação, aceleram o processamento e economizam energia.

Através da sua abordagem sistemática, a álgebra de Boole serve como uma ferramenta vital para entender e otimizar relações lógicas, tornando-se indispensável em áreas que vão da matemática à ciência da computação e engenharia.

Nos sistemas digitais, as portas lógicas são os componentes eletrónicos responsáveis por processar informações binárias de acordo com regras lógicas. Estas portas utilizam sinais de entrada binários, representados geralmente por níveis de tensão, com uma tensão positiva indicando o binário "1" e tensão nula indicando o binário "0". No entanto, a representação exata da tensão pode variar dependendo do design do circuito.

As portas lógicas mais comuns são a porta NOT, a porta AND e a porta OR, que correspondem às operações lógicas fundamentais. Existem também portas universais, como as portas NAND e NOR, que podem ser usadas para construir qualquer outro tipo de porta lógica, tornando-as essenciais no design de circuitos.

Além destas, portas lógicas especializadas, como XOR (OR Exclusivo) e XNOR (OR Exclusivo Negado), estendem a funcionalidade das operações booleanas. A porta XOR é especialmente útil em aplicações como adição binária e deteção de erros, enquanto a porta XNOR é usada para verificar se as entradas são idênticas.

Em resumo, as portas lógicas traduzem os princípios teóricos da álgebra booleana em ferramentas práticas para o design de circuitos digitais. Desde as portas básicas até as mais avançadas, estas componentes formam a espinha dorsal da eletrónica moderna, permitindo o desenvolvimento de sistemas de computação avançados.

Na álgebra de Boole, mintermos e maxtermos são conceitos fundamentais para a representação e análise de funções lógicas.

Um mintermo é expresso como uma conjunção (AND) de todas as variáveis de entrada, considerando as variáveis no seu estado original ou como complementos, dependendo

da combinação. Por outro lado, um maxtermo é descrito como uma disjunção (OR) das variáveis, seguindo o mesmo princípio. A relação entre mintermos e maxtermos é complementar, sendo definida pelas leis de De Morgan, que estabelecem que a negação de um mintermo equivale ao maxtermo correspondente, e vice-versa. Assim, para cada linha da tabela da verdade, o mintermo e o maxtermo associados descrevem, respetivamente, as condições de verdade e falsidade da função.

Estes conceitos são essenciais para as representações sob a forma canónica de funções lógicas: a soma de produtos (baseada na soma dos mintermos para os quais o valor da função é 1) e o produto de somas (baseado no produto dos maxtermos para os quais o valor da função é 0). Estas formas permitem descrever qualquer função booleana de forma sistemática e são amplamente aplicadas no design de circuitos lógicos digitais.

O uso de ferramentas como os Mapas de Karnaugh (K-maps) facilita a simplificação das expressões booleanas, agrupando mintermos ou maxtermos adjacentes para reduzir a complexidade do circuito. De acordo com a minimização de Karnaugh, a tabela da verdade de uma determinada expressão booleana é organizada numa grelha, onde cada célula representa um mintermo. O objetivo é agrupar os 1s (ou os 0s, no caso de maxtermos) adjacentes. Cada grupo representa um termo simplificado na expressão booleana.

Este método de simplificação de expressões booleanas utiliza a ordenação em código Gray para dispor os valores das variáveis de entrada, de modo que as células adjacentes diferem apenas por um dígito binário (bit). Esta organização facilita a identificação de 1s ou 0s adjacentes, tornando mais fácil agrupá-los. Ao minimizar o número de grupos e o número de literais dentro de cada grupo, o K-map gera uma expressão simplificada que pode ser implementada de forma eficiente em circuitos digitais. Este método oferece uma forma visual e sistemática de simplificar funções booleanas, levando a circuitos otimizados, com menos portas e entradas.

A álgebra de Boole tem um papel central na lógica digital, sendo essencial para o design e otimização de circuitos e sistemas computacionais. Particularmente nos dois tipos principais de circuitos digitais: os circuitos combinacionais e os circuitos sequenciais.

Os circuitos combinacionais são um tipo de circuito digital em que a saída depende exclusivamente das entradas atuais, sem considerar entradas ou saídas passadas. Estes circuitos, conhecidos como circuitos sem estado, funcionam inteiramente com base na lógica booleana, o que significa que, para um dado conjunto de entradas, a saída é previsível e determinística. São frequentemente utilizados em várias operações, incluindo tarefas aritméticas, processamento de dados e sistemas de controlo.

Uma característica importante dos circuitos combinacionais é que podem ter múltiplas saídas, cada uma correspondente a uma função booleana diferente, derivada diretamente das entradas. O comportamento destes circuitos pode ser representado através de tabelas da verdade e diagramas lógicos. Embora diferentes circuitos possam ser projetados para produzir a mesma tabela da verdade, a estrutura interna ou configuração do circuito pode variar.

Nos sistemas digitais, os somadores são circuitos combinacionais projetados para realizar a adição de números binários. Os somadores mais simples são o *half adder* e o *full adder*.

O *half adder* é um circuito simples que soma dois bits. É composto por duas portas lógicas: uma porta XOR (OR exclusivo) e uma porta AND. A porta XOR calcula a soma, gerando 1 quando exatamente um dos bits de entrada é 1. Já a porta AND calcula o *carry-out* (transporte), que é 1 apenas quando ambos os bits de entrada são 1. Embora o *half adder* seja útil para adições simples, ele é limitado, pois não consegue lidar com um *carry-in* (transporte de entrada) proveniente de uma adição anterior.

Para resolver esta limitação, foi desenvolvido o *full adder*. O *full adder* recebe três entradas: dois bits a serem somados e um *carry-in* ($C_{in}$ ) da adição anterior. Ele gera uma soma ($S$) e um *carry-out* ($C_{out}$), que pode ser transmitido para a próxima fase da adição. No *full adder*, a soma é calculada de maneira similar ao *half adder*, mas o *carry-out* é determinado com base nas três entradas: se duas ou mais entradas forem 1, o *carry-out* será 1.

Na prática, a adição binária é realizada encadeando vários *full adders* para somar números com múltiplos bits. A primeira fase de adição pode utilizar um *half adder* ou um *full*

*adder* com um $C_{in} = 0$, enquanto as fases subsequentes utilizam *full adders*, onde o $C_{out}$ de uma fase é passado como $C_{in}$ para a fase seguinte.

Assim como a adição é uma operação fundamental nos sistemas digitais, a subtração também pode ser realizada através de circuitos combinacionais. Um método comum de subtração binária é o uso do complemento para 2, que converte a subtração numa adição. No entanto, também podem ser projetados *full subtractors* e *half subtractor*es para realizar diretamente a subtração binária.

Um *half subtractor* recebe duas entradas binárias: o minuendo ($x$) e o subtraendo ($y$). São gerados dois valores: a diferença ($D$) e o transporte ($B$). O *half subtractor* funciona invertendo o subtraendo (usando uma porta NOT) e, em seguida, utilizando uma porta AND para calcular o transporte. Ao mesmo tempo, uma porta XOR calcula a diferença entre o minuendo e o subtraendo.

No entanto, um *half subtractor* sozinho não consegue realizar subtrações de múltiplos bits. Para lidar com números com mais de um bit, são usados *full subtractors*. Um *full subtractor* expande a função de um *half subtractor* ao incluir uma entrada *borrow-in* ($B_{in}$), permitindo-lhe lidar com casos em que a fase anterior de subtração gerou um transporte. O *full subtractor* recebe o minuendo, o subtraendo e o $B_{in}$ como entradas e gera uma diferença e um *borrow-out* ($B_{out}$), que é transmitido para o próximo subtrator na sequência.

Nos circuitos digitais, uma das principais distinções entre circuitos combinacionais e sequenciais é a presença de memória. Os circuitos combinacionais não possuem memória, i.e., a sua saída depende exclusivamente das entradas atuais e muda imediatamente quando essas entradas são alteradas. Como resultado, estes circuitos não retêm informações sobre entradas anteriores. Em contraste, os circuitos sequenciais incorporam elementos de memória, como latches e flip-flops, que lhes permitem reter informações sobre estados anteriores. Os circuitos sequenciais, portanto, dependem não apenas das entradas atuais, mas também dos estados passados, possibilanto operações mais complexas.

Os circuitos sequenciais podem ser classificados em síncronos (*flip-flops*) e assíncronos (*latches*). Os circuitos síncronos utilizam um sinal de relógio comum para

atualizar os seus estados em intervalos regulares, garantindo uma temporização previsível. Nestes circuitos, as atualizações de estado são acionadas pelo pulsar do relógio. Por outro lado, os circuitos assíncronos atualizam os seus estados com base nas mudanças das entradas, sem a necessidade de um sinal de relógio global.

Os flip-flops são dispositivos ativados no flanco (ascendente ou descendente), alterando o seu estado quando há uma transição no sinal de relógio (de *Low* para *High* ou vice-versa). Isto torna os flip-flops ideais para aplicações que requerem mudanças de estado sincronizadas, como o armazenamento de memória em sistemas sequenciais. O SR flip-flop tem duas entradas, Set (S) e Reset (R), e é projetado para ser acionado por um sinal de relógio.

Por outro lado, os latches são dispositivos sensíveis ao nível, ou seja, as suas saídas podem mudar continuamente enquanto o sinal de ativação estiver ativo, respondendo imediatamente às mudanças nas entradas, o que os torna adequados para aplicações que requerem atualizações contínuas. À semelhança do SR flip-flop, o SR latch também tem duas entradas, Set (S) e Reset (R), no entanto é sensível ao nível e não acionado por um sinal de relógio. Num SR latch com portas NOR, quando S é *High*, o latch coloca a sua saída em H*igh* (1), e quando R é *High*, o latch coloca a sua saída em *Low* (0).

Além da já estabelecida ligação entre a álgebra de Boole e os circuitos digitais, novas aplicações desta lógica foram mais recentemente sugeridas. Em 2023 Pertersen *et al.* publicaram uma arquitetura de redes neuronais baseadas em portas lógicas ou *Logic Gate Neural Networks* (LGNs).

Dentro do vasto campo da inteligência artificial (AI), destaca-se o *Machine Learning* (ML), que se foca em capacitar sistemas para tomarem decisões baseadas em dados, através da construção e aplicação de modelos matemáticos. Dentro do ML, o *Deep Learning* tem ganho grande relevância, ao treinar as chamadas *Deep Neural Networks* com grandes conjuntos de dados. Esta abordagem tornou-se fundamental em aplicações como o processamento de linguagem natural (NLP), *computer vision* e reconhecimento de voz.

Os modelos de ML dividem-se geralmente em três categorias: aprendizagem supervisionada, aprendizagem não supervisionada e *reinforcement learning*. Na primeira,

utilizam-se dados identificados (*labeled*) para treinar os modelos, permitindo-lhes mapear entradas para saídas correspondentes. Por exemplo, um modelo treinado para classificar géneros musicais usaria amostras de áudio identificadas para aprender a relação entre o áudio (input) e o género (output). Por outro lado, a aprendizagem não supervisionada lida com dados não identificados, tentando identificar padrões ou estruturas sem saídas predefinidas.

Uma arquitetura central no *deep learning* são as redes neuronais (NNs), compostas por camadas de *perceptrons* (neurónios artificiais). O *Perceptron*, introduzido por Rosenblatt em 1958, tomava decisões binárias com base na soma ponderada dos inputs $\left(\sum_j w_j x_j = \boldsymbol{w} \cdot \boldsymbol{x}\right)$ e num limiar pré-definido $(\boldsymbol{w} \cdot \boldsymbol{x} \leq threshold\ value \leftrightarrow \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0)$. Quando vários perceptrons são interligados, formam uma NN capaz de decisões mais complexas.

Numa NN, as camadas (*layers*) são componentes essenciais. Uma NN simples de tipo *feed forward*, ou *Feed Forward Neural Network* (FFN) contém três camadas principais: camada de entrada (*input layer)*, camadas ocultas *(hidden layers)* e camada de saída (*output layer*). Cada neurónio numa camada conecta-se a neurónios na camada seguinte, e a saída de cada neurónio (output) é determinada pela aplicação de uma função de ativação a uma soma ponderada das suas entradas (input).

As FFNs são modelos matemáticos $(f)$ que mapeiam entradas multivariadas $\boldsymbol{x}$ para saídas $\boldsymbol{y}$ através de um conjunto de parâmetros $\boldsymbol{\phi}$. Por exemplo, $\boldsymbol{y} = f[\boldsymbol{x}, \boldsymbol{\phi}] = \phi_0 + \phi_1 a[\theta_{10} + \theta_{11} x] + \phi_2 a[\theta_{20} + \theta_{21} x] + \phi_3 a[\theta_{30} + \theta_{31} x]$. Esta rede contém três funções lineares, cada uma aplicada a uma função de ativação $a[\cdot]$ (por exemplo, a ReLU), sendo depois ponderadas e somadas para calcular o output.

A função de ativação ReLU $\left(ReLU[z] = \begin{cases} 0, & z \leq 0 \\ z, & z > 0 \end{cases}\right)$ é amplamente utilizada, pois devolve o valor da entrada se for positivo e zero caso contrário. Os parâmetros $\boldsymbol{\phi}$ controlam a forma da função que mapeia inputs para outputs.

Como o objetivo de treinar uma NN é determinar os valores dos parâmetros que minimizam uma função de custo ou *loss function* $(L[\boldsymbol{\phi}])$, a qual quantifica a diferença entre as previsões do modelo e os valores reais, utiliza-se frequentemente o método do gradiente

descendente, uma técnica de otimização que ajusta os parâmetros com base no gradiente da função de custo em relação a cada parâmetro. A regra de atualização na descida do gradiente segue $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha \frac{\partial L}{\partial \boldsymbol{\phi}}$. Onde $\alpha$ é a *learning rate*, que controla a dimensão de cada ajuste. Este processo repete-se até que a função de custo atinja um mínimo, momento em que o gradiente se torna zero, indicando que já não é possível fazer mais melhorias significativas.

Uma variante amplamente usada é o gradiente descendente estocástico, que introduz um elemento de aleatoriedade. Em vez de calcular os ajustes com base em todo o conjunto de dados, este método utiliza apenas um subconjunto aleatório (*batch*) em cada iteração, tal que $\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$, onde $\ell_i$ é o resultado da função de custo na observação $i$, e $\mathcal{B}_t$ é um conjunto que contém os índices dos pares de entrada/saída no *batch* atual. Este método reduz a carga computacional e, ao mesmo tempo, ajuda o algoritmo a escapar de mínimos locais, melhorando a capacidade de generalização do modelo.

No entanto, durante o treino, podem surgir dois problemas: *underfitting* e *overfitting*. O primeiro ocorre quando o modelo é demasiado simples para captar os padrões subjacentes dos dados. Já o segundo acontece quando o modelo é demasiado complexo, captando ruído nos dados e falhando em generalizar para novas observações. Estes problemas podem ser mitigados ajustando a arquitetura do modelo e os hiperparâmetros, como o número de camadas e neurónios, ou usando conjuntos de validação para monitorizar o desempenho durante o treino.

Uma vez treinado e avaliado num conjunto de teste, o modelo pode ser implementado para uso prático. Selecionar os hiperparâmetros adequados e estratégias de treino eficazes é crucial para alcançar um desempenho ideal e garantir que o modelo generalize bem para aplicações no mundo real.

As LGNs, sugeridas por Petersen *et al.* (2023), apresentam-se como uma alternativa aos modelos convencionais de NNs. Nestas redes, os neurónios são definidos por conjuntos de portas lógicas binárias, como AND, NAND ou NOR. Ao contrário das redes neuronais tradicionais, cujas funções que representam são diferenciáveis, as LGNs baseiam-se em funções lógicas intrinsecamente não diferenciáveis, o que coloca desafios à otimização

baseada em gradientes durante o treino. Para superar esta limitação, Petersen *et al.* (2023) propuseram um método de relaxamento, transformando as ativações binárias em valores probabilísticos entre 0 e 1.

Na versão diferenciável das LGNs, cada neurónio é parametrizado por uma distribuição de probabilidade sobre 16 possíveis funções lógicas. Durante o treino, estas distribuições são aprendidas, e, no final, seleciona-se a função com maior probabilidade para cada neurónio. O resultado é uma rede esparsa, onde cada neurónio possui apenas duas entradas. Em vez de aprender pesos, as LGNs determinam a função lógica ideal para cada neurónio, tornando-as mais eficientes em termos computacionais e de memória. Além disso, ao executar operações binárias em vez de cálculos com números em vírgula flutuante, estas redes são mais rápidas e adequadas para ambientes com restrições de energia, como sistemas embutidos.

Neste trabalho a aplicabilidade prática das LGNs foi testada com um conjunto de imagens de radiografias pulmonares para diagnóstico de pneumonia, composto por imagens classificadas como "Normais" ou "Opacas". As imagens foram pré-processadas e redimensionadas para $40 \times 40$ pixéis. O conjunto de dados selecionado já se encontrava dividido em subconjuntos de treino, validação e teste.

A arquitetura das LGNs foi implementada em *PyTorch*, utilizando o repositório disponibilizado pelos autores do artigo *Deep Differentiable Logic Gate Networks*, e treinada com s*tochastic gradient descent* e optimizador Adam. Para as LGNs, foi necessário transformar os píxeis das imagens em entradas binárias. Foram utilizados um tamanho de *batch* de 4 e uma *learning rate* de 0,0001, com *early-stopping* para evitar *overfitting*. Após o treino, o modelo alcançou uma exatidão de teste de 89,9% ao fim de 46 *epochs*, com um tempo de treino total de 705 segundos.

Esta abordagem, baseada nas LGNs diferenciáveis, demonstrou o potencial das LGNs para alcançar eficiência computacional e de memória, mantendo uma elevada exatidão em tarefas de classificação binária. A capacidade de aprender e otimizar funções lógicas torna as LGNs uma alternativa promissora para aplicações em tempo real e sistemas embutidos.

O estudo analisou também a implementação de FFNs utilizando o mesmo conjunto de dados e a função de ativação ReLU numa arquitetura padrão de camadas totalmente conectadas, alcançando-se uma exatidão de 80,4% em 13 *epochs*, com um tempo de treino de 191 segundos. Para este treino não foi necessário transformar cada píxel em vetores binários, já que esta arquitetura processa diretamente vetores de vírgula flutuante.

Apesar de ambos os modelos apresentarem um número semelhante de parâmetros, as LGNs mostraram maior capacidade para capturar relações complexas nos dados, embora exijam mais recursos computacionais durante o treino. As FFNs, por outro lado, treinaram mais rapidamente, mas com menor desempenho.

O estudo destacou as vantagens das LGNs em termos de eficiência computacional e de memória, posicionando-as como uma solução promissora para aplicações em tempo real e de baixo consumo. Embora o estudo tenha sido realizado com ajustes mínimos de hiperparâmetros, trabalhos futuros poderão explorar técnicas de otimização adicionais, como busca por hiperparâmetros e regularização, para melhorar o desempenho e a capacidade de generalização do modelo.

# Contents

# List of Tables

# List of Figures

# List of Abbreviations and Acronyms

| | |
|---|---|
| Adam | Adaptive Moment Estimation |
| AI | Artificial Intelligence |
| ALU | Arithmetic Logic Unit |
| CPU | Central Processing Unit |
| FA | Full Adder |
| FFN | Feed Forward Neural Network |
| FS | Full Subtractor |
| GPU | Graphics Processing Unit |
| HA | Half Adder |
| HS | Half Subtractor |
| HTML | Hypertext Markup Language |
| IC | Integrated Circuit |
| LGN | Logic Gate Neural Network |
| LSB | Least Significant Bit |
| MAPE | Mean Average Percentage Error |
| ML | Machine Learning |
| MLP | Multilayer Perceptron |
| MNIST | Modified National Institute of Standards and Technology |
| MSB | Most Significant Bit |
| MSE | Mean Squared Error |
| NLP | Natural Language Processing |
| NN | Artificial Neural Network |
| ReLU | Rectified Linear Unit |
| SNN | Shallow Neural Network |
| SR | Set-Reset |

# INTRODUCTION

The rapid evolution of modern computing is deeply rooted in foundational mathematical tools that govern the logic of digital systems. Among these, the study of Boolean algebra plays a central role in shaping the theoretical and practical aspects of computation. The diversity of number systems provides essential frameworks for representing and processing data in various contexts, with the binary system standing out due to its compatibility with electronic devices, thus forming the backbone of digital computation. At the same time, Boolean algebra, with its operations and properties such as the basic logic functions (AND, OR, and NOT), truth tables, and laws like De Morgan's, serves as a mathematical pillar for digital logic and computation. It enables the logical modeling of digital circuits and computational processes, transforming abstract mathematical concepts into tangible, practical applications.

The objective of this dissertation is to explore Boolean algebra, focusing on its role in the evolution of digital circuits and their continuous relevance in the design of modern computational systems. More specifically, the research delves into how Boolean algebra's application in both combinational and sequential circuits, along with its innovative adaptation in emerging technologies like artificial intelligence (AI), continues to drive innovations in the digital era. The integration of Boolean principles with cutting-edge advancements in machine learning (ML), particularly in Logic Gate Networks (LGNs), underscores the ongoing significance of Boolean algebra, demonstrating its transformative impact in new areas.

The methodology employed in this study combines both theoretical analysis and practical experimentation. A comprehensive review of relevant literature provides the historical and mathematical context for Boolean algebra and its integration into modern digital circuits. The dissertation also incorporates empirical data and case studies, particularly from the application of LGNs in ML tasks, such as medical image classification. Through the analysis of an LGN model, this research examines the advantages and challenges of Boolean-based neural networks in terms of efficiency, memory usage, and computational cost.

This dissertation begins with a thorough analysis of Boolean algebra. The first chapter introduces numeral systems, with a focus on binary notation and its essential link to Boolean logic. The second chapter delves deeper into Boolean algebra, covering fundamental concepts, core theorems and Boolean identities, including De Morgan's laws, which are essential for algebraic simplification.

Building on this knowledge, the third chapter examines the role of logic gates in digital circuits, highlighting their importance as the fundamental components in digital operations. Several types of logic gates, such as NOT, AND, OR, NAND and so on, are discussed in detail, along with how these gates can be combined to create more complex functions.

On its turn, the fourth chapter covers digital circuit modelling, emphasizing the simplification and optimization of Boolean expressions. Key concepts like minterms and maxterms are introduced, along with methods for expressing Boolean functions using sum of products and product of sums formats. This chapter concludes with Karnaugh Maps, a tool used to reduce the number of required gates, further optimizing circuit efficiency.

After this conceptual overview, the dissertation explores the classic applications of Boolean algebra within digital circuits. Chapter five focuses on combinational circuits, which are defined solely by input values and lack memory elements. Arithmetic circuits like adders and subtractors are examined in detail. On the other hand, chapter six is centered on sequential circuits, which differ from combinational circuits by incorporating memory, allowing outputs to depend on prior states as well as current inputs. This chapter explores key components like latches and flip-flops, and it also discusses binary counters as a practical example.

After exploring the traditional applications of Boolean algebra, this dissertation turns its focus to the innovative crossing between Boolean algebra and NNs. Chapter seven is dedicated to the introduction of ML, particularly deep learning, as a novel application area for Boolean principles. It provides an overview of essential ML concepts, classic NN architectures, as well as their training process. Building on these concepts, chapter eight presents an innovative architecture known as LGNs, a type of NN inspired by Boolean principles.

This approach, initially proposed by Petersen *et al.* (2023), integrates logic functions directly into the NN architecture. The main innovation, introduced by these authors, is the adaptation of these networks into differentiable versions, which allows the use of gradient-based optimization algorithms.

This chapter stands out from the rest, as it presents the practical experiment of this dissertation by implementing a LGN to a selected dataset for an image classification exercise and analyzing the results. In parallel an FFN will also be implemented to the same dataset and with approximately the same number of trainable parameters and subsequently analyzed.

Finally, this dissertation proposes a hybrid approach that integrates theory and practice, combining the principles of Boolean algebra with the latest innovations in deep learning. It is believed that the integration of these areas can offer new perspectives for the development of more efficient NNs, with enhanced performance in terms of computational optimization and generalization capacity.

In conclusion, this research aims to underscore the critical role that Boolean algebra continues to play in the advancement of both hardware and software technologies. By examining its historical significance and contemporary applications, the dissertation illustrates how the convergence of mathematical logic and technological innovation propels the development of more efficient, reliable, and advanced computational systems, paving the way for future breakthroughs in artificial intelligence and beyond.

# CHAPTER 1 : NUMERAL SYSTEMS

## 1.1 Introduction to Numeral Systems

This first chapter starts by introducing numeral systems, providing an essential understanding of the different ways numbers are represented in computing. The binary system, which is particularly relevant for digital systems, is examined, as well as the decimal system. This chapter establishes the groundwork for understanding how digital circuits operate and how numbers are processed within them.

Number systems form the foundation of mathematical operations and computations, providing a framework for representing and manipulating quantities. Whether it's counting, measuring, or performing complex mathematical operations, numeral systems offer a structured framework for communicating numeral information. Different bases and systems provide flexibility in expressing numeral concepts and solving mathematical problems.

Common systems include the decimal, binary, octal, and hexadecimal systems, each with its own base and symbols. The binary system is crucial in digital electronics and computer programming, as it aligns with the way computers process information. It is also closely related to Boolean algebra, which underpins digital circuit design. The octal and hexadecimal systems are often used for compact representations of binary data, making them important tools in computing.

## 1.2 Decimal System

The decimal system, also known as the base 10 system, is perhaps the one most familiar to us, humans. This system employs ten symbols, digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), and it is considered a positional system where the value of a digit depends on its position within a number, i.e., “depending on its position in the string, each digit has an associated value of an integer raised to the power of 10” (Mano & Kime, 2014: 13). Taking the example of the number 247, the '2' represents 200, the '4' represents 40, and the '7' represents 7, this is made evident in Figure 1.1 below.

**Figure 1.1:** Representation in the decimal system.

$$\mathbf{247} = \mathbf{2} \times 10^2 + \mathbf{4} \times 10^1 + \mathbf{7} \times 10^0 = \mathbf{2}00 + \mathbf{4}0 + \mathbf{7}$$

Convention says to write only the digits and deducing the corresponding powers of 10 from their positions, increasing from right to left. The decimal system is known as a base (or *radix)* 10 system, because "the coefficients are multiplied by powers of 10 and the system uses 10 distinct digits" (Mano & Kime, 2014: 13). This system is the most used in everyday life. It is used for counting money, measuring distances, and in various other practical applications. Most of our day-to-day calculations, such as shopping expenses or budgeting, are done in the decimal system.

### 1.3 Binary System

The binary system is a base (or *radix*) 2 system that, as the name suggests, only uses two symbols: 0 and 1. Although humans typically find it more intuitive to work with the decimal system, it would be very difficult and expensive to design electronic equipment that could work with ten different voltage levels. On the other hand, it has been proven very easy to design electronic circuits that operate with only two voltage levels: 0 and 1. This explains why the binary system plays such a crucial role in computer science and engineering.

In fact, the binary system aligns itself perfectly with the underlying electronic nature of computing devices. For example, electronic devices like transistors, which form the building blocks of modern computer circuits, have two states: conductive and non-conductive, fitting in very well with the binary system, where 0 typically represents the non-conductive state, and 1 usually represents the conductive state. Additionally, this system is very reliable, as the clear distinction between 0 and 1 helps reduce the chances of misinterpretation or corruption of data during transmission.

Despite only using two possible digit values, the binary system can still be used to represent any quantity that can be represented through the other numeral systems, it would only take a greater number of binary digits to do so.

This system is also a positional-value system, where each binary digit has its own weight expressed as a power of 2. Taking the number 101.1 as an example, it is shown in (1.1) the equivalent decimal value.

$$(101.1)_2 = (1 * 2^2) + (0 * 2^1) + (1 * 2^0) + (1 * 2^{-1}) = (5.5)_{10} \qquad (1.1)$$

It is worth noting that, whenever more than just one system is employed, the subscripts (2 for binary, 10 for decimal and so on) should be used to assure clarity. This is made evident in Figure 1.2 below.

**Figure 1.2:** Different notations between the binary and decimal systems.

$$\underbrace{\mathbf{10111_2 \neq 10111_{10}}}$$

$$\mathbf{10111_2} = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = \mathbf{23_{10} \neq 10111_{10}}$$

Digital information, whether text or images, is represented in binary code. Computers routinely convert between binary and decimal to interact with users. Humans input and receive data in decimal format, while computers convert their internal binary data to decimal for user readability.

The use of binary at the machine level is fundamental to the operation of digital computers. Taking the previous example given in Figure 1.2 above, it is possible to see the conversion from binary to decimal. The conversion from decimal to binary is a little more complex. It starts with writing down the decimal number, for example 10111. Then, this number is divided by 2 to get the digits from the remainders (either 0 or 1). The division of the quotient by 2 continues until the integer quotient becomes 0, writing down the remainders each time. The sequence of remainders is read from bottom to top to obtain the binary equivalent. This process is illustrated in Table 1.1 below.

**Table 1.1:** Conversion from decimal to binary.

| Division by 2 | Quotient | Remainder | Bit |
|---|---|---|---|
| 10111/2 | 5055 | **1** | 0 |
| 5055/2 | 2527 | **1** | 1 |
| 2527/2 | 1263 | **1** | 2 |
| 1263/2 | 631 | **1** | 3 |
| 631/2 | 315 | **1** | 4 |
| 315/2 | 157 | **1** | 5 |
| 157/2 | 78 | **1** | 6 |
| 78/2 | 39 | **0** | 7 |
| 39/2 | 19 | **1** | 8 |
| 19/2 | 9 | **1** | 9 |
| 9/2 | 4 | **1** | 10 |
| 4/2 | 2 | **0** | 11 |
| 2/2 | 1 | **0** | 12 |
| 1/2 | 0 | **1** | 13 |
| $= 10011101111111_2$ | | | |

## 1.4 Octal and Hexadecimal Systems

In addition to the decimal and binary systems, there are other numeral systems commonly used in computer science and mathematics, such as the octal and hexadecimal systems. These systems are particularly useful for representing large binary numbers in a more compact form. For a detailed explanation of the octal and hexadecimal systems, please refer to Appendix I.

## 1.5 Numeral Systems and Boolean Algebra

As previously mentioned, hardware elements known as digital circuits operate on binary information. These circuits are implemented using transistors and interconnections in

complex semiconductor devices known as integrated circuits[1] (ICs). The fundamental building block of these circuits is called a logic gate. To simplify the design process, the electronic circuits are modeled with transistors as logic gates (Mano & Kime, 2014). This approach allows designers to abstract away the internal electronics of individual gates, focusing solely on their external logical properties.

Each logic gate performs a specific logical function, with their outputs connecting to other gates to create digital circuits. These functions are described by Boolean algebra, which is fundamental for designing and analyzing digital circuits. Boolean algebra will be explored in more detail in the next chapter.

## 1.6 Conclusion

In conclusion, numeral systems are fundamental to both mathematical theory and practical computation. From the familiar decimal system to the binary system, each plays a vital role in representing and manipulating numerical data. Understanding these systems provides a solid foundation for more advanced topics in computer science and digital electronics. Furthermore, the relationship between numeral systems and Boolean algebra highlights the importance of binary logic in modern computing. Boolean algebra serves as the backbone of digital circuit design and programming, emphasizing the practical application of these theoretical concepts in technology and computation.

[1] While Boolean algebra provides the theoretical framework for circuit design, practical implementation is achieved through ICs. Logic gates are not sold individually but are packaged within ICs, which are then combined to form the intended circuits. Early ICs, known as Small-Scale Integration chips, contained up to 100 electronic components per chip. Technological advancements have dramatically increased this capacity, leading to Ultra Large-Scale Integration chips. In 1965, Gordon Moore, co-founder of Intel®, predicted that the number of transistors on a microchip doubles approximately every two years, leading to increased computational power and decreased relative cost. This observation, known as Moore's Law, has guided the semiconductor industry for decades, driving rapid technological advancements and shaping modern computing.

# CHAPTER 2 : BOOLEAN ALGEBRA

## 2.1 Introduction to Boolean Algebra

This second chapter dives deeper into Boolean algebra, the mathematical system that uses binary variables and logical operations to model and simplify logical expressions. Boolean algebra is fundamental to the design and optimization of digital circuits, which are built from logic gates. Key identities, properties, and simplification techniques, including De Morgan's laws, which play a crucial role in reducing the complexity of logical expressions, will be explored.

In 1853 George Boole introduced a systematic treatment of logic, developing an algebraic system known today as symbolic logic or Boolean algebra. Boole authored an essay titled *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities*, wherein he formalized rules governing the relationships between mathematical quantities confined to binary values: true or false, often represented as 1 or 0 respectively.

This mathematical framework later became recognized as Boolean algebra and found application in the 1930s when the German engineer Konrad Zuse utilized it for his Z1 calculating machine (Mano & Kime, 2014).

Additionally, it played a crucial role in the development of the first digital computer in the late 1930s, by the American physicist John Atanasoff and his graduate student (Mano & Kime, 2014). In the 1930-40s, British mathematician Alan Turing and American mathematician Claude Shannon independently recognized the suitability of binary logic for advancing digital computer technology (Mano & Kime, 2014). Between 1944-45, mathematician John von Neumann proposed the use of binary arithmetic to store programs in computers (Mano & Kime, 2014).

Boolean algebra, a mathematical structure rooted in the logic of binary variables, serves as a cornerstone in various scientific and technological disciplines. This chapter delves into the essence of Boolean algebra, exploring its theorems and properties.

## 2.2 Definition and Fundamental Concepts

Boolean algebra is a mathematical structure that deals with binary variables and logical operations. At its core, Boolean algebra operates in a binary system where variables take on one of two values: 1 or 0. The fundamental concepts include Boolean variables, logical operations (AND, OR, NOT) and Boolean expressions.

Boolean variables represent the binary states, while logical operations define the relationships between these states, and expressions serve as concise representations of complex logical relationships.

There are three basic operations:

- **NOT Operation** ($\bar{x}$ or $x'$)**:** negates the input. If $x$ is TRUE, $\bar{x}$ is FALSE and vice versa.
- **AND Operation** ($x \wedge y$ or $x * y$ or $xy$): results in TRUE only if both inputs are TRUE. $x \wedge y$ is TRUE only when $x$ is TRUE and $y$ is also TRUE.
- **OR Operation** ($x \vee y$ or $x + y$): results in TRUE if at least one input is TRUE. $x \vee y$ is TRUE when only $x$ is TRUE, when only $y$ is TRUE, or when both $x$ and $y$ are TRUE.

Any logic expression composed of NOT, AND or OR operation is considered a Boolean function. These typically have one or more input values and yield a result in the range [0,1].

A Boolean operator can be described using a table that shows the tabular relationship between the input values and the results of a specific operator or function on the input variables. These are called truth tables.

The NOT operator, as already mentioned, is the negation of an input. Both "$\bar{x}$" and "$x'$" are read as *"not x"*. The logic NOT function acts as a single input inverter, changing the input of a value "1" to an output of value "0" and vice versa. The truth table shown in Table 2.1 below demonstrates the behavior of this operator.

**Table 2.1:** The truth table for the NOT function.

| INPUTS | OUTPUTS |
|---|---|
| $\boldsymbol{x}$ | $\boldsymbol{\overline{x}}$ |
| 0 | 1 |
| 1 | 0 |

The AND operator is also known as a Boolean product or multiplication and the Boolean expression "$xy$" corresponds to "$x \wedge y$" or to "$x * y$" and it reads "*x and y"*. According to the logic AND function, two or more events need to happen simultaneously for an output action to take place. However, the order in which these actions occur is irrelevant as it does not impact the result. The truth table shown in Table 2.2 denotates the behavior of this operator.

Lastly, the OR operator, commonly known as a Boolean sum or addition, is represented by the expressions "$x \vee y$" or "$x + y$", both read as "*x or y"*. The logic OR function states that an output action will take place if one or more events occur, but, as before, the order at which they occur is not relevant as it does not impact the result (T.P., 2018). This function is also known as "Inclusive-OR", contrasting to the "Exclusive-OR Function" that will be briefly described in Chapter 3. The truth table for the inclusive-OR is also shown in Table 2.2 below.

**Table 2.2:** The truth table for the AND and OR functions.

| INPUTS | | AND | OR |
|---|---|---|---|
| $\boldsymbol{x}$ | $\boldsymbol{y}$ | $\boldsymbol{xy}$ | $\boldsymbol{x + y}$ |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

According to the rule of precedence for Boolean operators, the operator NOT has priority, followed by AND, and at last OR. The truth table for $F(x, y, z) = x + \bar{y}z$ is given in Table 2.3 below as an example of this.

**Table 2.3:** The truth table for $x + \bar{y}z$.

| INPUTS | | | | | OUTPUTS |
|---|---|---|---|---|---|
| $x$ | $y$ | $z$ | $\bar{y}$ | $\bar{y}z$ | $x + \bar{y}z$ |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |

Source: Yang, 2020:3.

## 2.3 Boolean Algebraic Identities and Properties

In the world of equations, a statement that holds true for every value of its variable or variables is known as an identity (Yang, 2020). A property (or law) is considered a type of mathematical identity, that outlines the relationships between distinct variables within a numeral system (Yang, 2020). The elegance and power of Boolean algebra emerges from a set of identities and properties that govern the manipulation of Boolean expressions.

Basic identities such as idempotence, commutativity, and distributivity form the foundation, ensuring consistency and predictability in logical operations. Properties like De Morgan's laws play a crucial role in simplifying complex expressions, offering a systematic

approach to transforming logical statements. According to the literature the basic laws of Boolean algebra are represented in Table 2.4 below.

**Table 2.4:** Basic laws of Boolean algebra.

| BASIC LAW | LOGICAL PRODUCT (AND FORM) | LOGICAL SUM (OR FORM) |
|---|---|---|
| **Identity** | $1x = x$ | $0 + x = x$ |
| **Null or Dominance** | $0x = 0$ | $1 + x = 1$ |
| **Idempotent** | $xx = x$ | $x + x = x$ |
| **Inverse** | $x\bar{x} = 0$ | $x + \bar{x} = 1$ |
| **Commutative** | $xy = yx$ | $x + y = y + x$ |
| **Associative** | $(xy)z = x(yz)$ | $(x + y) + z = x + (y + z)$ |
| **Distributive** | $x + yz = (x + y)(x + z)$ | $x(y + z) = xy + xz$ |
| **Absorption** | $x(x + y) = x$ | $x + xy = x$ |
| **Double Complement** | $\bar{\bar{x}} = x$ | |
| **De Morgan's** | $\overline{xy} = \bar{x} + \bar{y}$ | $\overline{x + y} = \bar{x}\bar{y}$ |
| **Adjacency** | $(x + y)(x + \bar{y}) = x$ | $xy + x\bar{y} = x$ |

Adapted from: Yang, 2020; Mano & Kime, 2014.

Taking the expression given previously in Table 2.3 above and applying the distributive law, it can be further manipulated, obtaining its equivalent $x + \bar{y}z \Leftrightarrow (x + \bar{y}) \cdot (x + z)$.

Furthermore, the duality principle in Boolean algebra states that any theorem or identity in Boolean algebra remains valid if, simultaneously:

- Every "AND" (•) is replaced by an "OR" (+).
- Every "OR" (+) is replaced by an "AND" (•).
- Every "0" is replaced by a "1".
- Every "1" is replaced by a "0".

In simpler terms, it means that for any statement or expression in Boolean algebra, if ANDs are interchanged with ORs and 0s are replaced with 1s and 1s with 0s, a valid statement or expression known as its *dual* will be obtained, i.e., $x \cdot 1 = x$ is the *dual expression* of $x + 0 = x$ (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). This principle is quite powerful and useful in simplifying Boolean expressions and proving theorems, allowing one to derive new expressions or identities from known ones by simply applying the transformations described above (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

De Morgan's laws are a pair of fundamental rules in Boolean algebra that describe the relationship between logical operations (AND, OR) and their complements (NOT). These laws are named after the mathematician and logician Augustus De Morgan.

## 2.4 De Morgan's Laws

Augustus De Morgan was a British mathematician and logician who made significant contributions to the fields of mathematics and symbolic logic. More concretely, De Morgan worked on the development of mathematical logic and set theory. His best-known work is perhaps the De Morgan's laws, that prove highly beneficial when simplifying expressions that involve the inversion of a product or sum of variables. These laws describe the relationships between the logical operations of negation (NOT), conjunction (AND), and disjunction (OR).

According to De Morgan's first law, the negation of a conjunction (AND) is the same as the disjunction (OR) of the negations, i.e., "the complement of a logical product equals the logical sum of the complements" (2.1) below (Bhagat:47). This law states that inverting the AND product of two variables is equivalent to individually inverting each variable and then performing a logical OR operation. In set theory, this law states that the complement of the intersection of two sets is the same as the union of their complements (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

$$\overline{xy} = \bar{x} + \bar{y} \qquad (2.1)$$

On the other hand, De Morgan's second law states that the negation of a disjunction is the conjunction of the negations, i.e., "the complement of a logical sum equals the logical product of the complements" (2.2) (Bhagat:47). This law asserts that inverting the OR sum of two variables is equivalent to individually inverting each variable and subsequently ANDing the inverted variables (Bhagat). In set theory, this law states that the complement of the union of two sets is the same as the intersection of their complements (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

$$\overline{x + y} = \bar{x}\,\bar{y} \tag{2.2}$$

De Morgan's laws, validated through exhaustive examination of all possible combinations of $x$ and $y$, extend their applicability beyond single variables to situations where $x$ and/or $y$ denote expressions with multiple variables. When utilizing these laws to simplify an expression, one can systematically break an inverter sign at any point within the expression, replacing the operator at that point with its opposite, i.e., "+" is replaced by " $\cdot$ " and vice versa; and this process persists until the expression is ultimately simplified to a state where only single variables undergo inversion like shown in (2.3). Additionally, in this example the double complement law is also applied to further manipulate the expression.

$$\text{Example (1): } \overline{(x + yw) \cdot (c + de)} = \overline{x + yw} + \overline{c + de} = (\bar{x} \cdot \overline{yw}) + \left(\bar{c} \cdot \overline{de}\right)$$
$$= \bar{x} \cdot (\bar{y} + \bar{w}) + \bar{c} \cdot \left(\bar{d} + \bar{e}\right) = \bar{x}\bar{y} + \bar{x}\bar{w} + \bar{c}\bar{d} + \bar{c}\bar{e}$$

$$Example\ (2):\ \overline{x + \bar{y}.w} = \bar{x} \cdot \overline{\bar{y} \cdot w} = \bar{x} \cdot (\bar{\bar{y}} + \bar{w}) = \bar{x} \cdot (y + \bar{w}) = \bar{x}y + \bar{x}\bar{w} \tag{2.3}$$

De Morgan's laws are foundational principles in logic, establishing crucial equivalences and rules for transforming expressions with negations, conjunctions (AND), and disjunctions (OR). These laws are particularly useful when working with logic circuits and expressions as they provide a way to simplify complex Boolean expressions by expressing them in terms of complements and simpler operations. In Table 2.5 below it is demonstrated the De Morgan's laws through truth tables and in (2.4)[2] below it is shown the application of

[2] Example adapted from Mano & Kime, 2014.

these laws for simplification of a Boolean expression, putting negation at variable level (Mano & Kime, 2014).

$$\overline{a.\left(b+z.(x+\bar{a})\right)} = \bar{a} + \overline{b+z.(x+\bar{a})} = \bar{a} + \bar{b}.\overline{\left(z.(x+\bar{a})\right)}$$

$$= \bar{a} + \bar{b}.\left(\bar{z} + (\overline{x+\bar{a}})\right) = \bar{a} + \bar{b}.(\bar{z} + \bar{x}\cdot\bar{\bar{a}})$$

$$= \bar{a} + \bar{b}.(\bar{z} + \bar{x}.a) \qquad (2.4)$$

**Table 2.5:** Demonstration of the De Morgan's laws through truth tables.

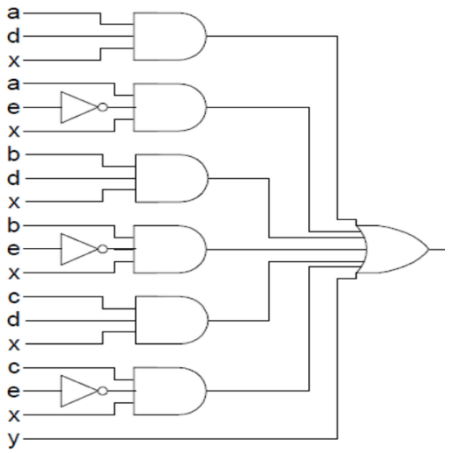
| $\overline{xy} = \bar{x} + \bar{y}$ | | | | | | |
|---|---|---|---|---|---|---|
| $x$ | $y$ | $xy$ | $\overline{xy}$ | $\bar{x}$ | $\bar{y}$ | $\bar{x}+\bar{y}$ |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| $\overline{x+y} = \bar{x}\cdot\bar{y}$ | | | | | | |
| $x$ | $y$ | $x+y$ | $\overline{x+y}$ | $\bar{x}$ | $\bar{y}$ | $\bar{x}\cdot\bar{y}$ |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

## 2.5 Algebraic Simplification of Boolean Expressions

The algebraic simplification of Boolean expressions entails many advantages. Namely, as it will be seen in the next chapters, it makes circuits less complex, reducing assembly errors; it speeds up signal transmission through computing circuits; and it lowers power consumption (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

Taking the example illustrated in Figure 2.1[3] below, the original expression can be simplified by applying Boolean laws and identities as shown in (2.5), arriving at the simplified circuit shown in Figure 2.2.

**Figure 2.1:** Example of a 2-level implementation circuit.




$$f = adx + a\bar{e}x + bdx + b\bar{e}x + cdx + c\bar{e}x + y =$$

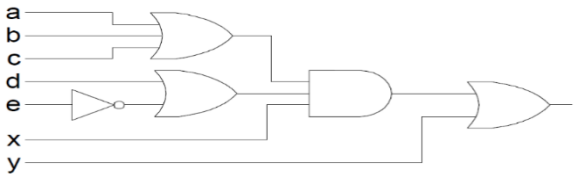
$$= (ad + a\bar{e} + bd + b\bar{e} + cd + c\bar{e})x + y$$

$$= \big((a + b + c)d + (a + b + c)\bar{e}\big)x + y =$$

$$= (a + b + c).(d + \bar{e}).x + y \qquad (2.5)$$

Adapted from: Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019.

**Figure 2.2:** Multilevel implementation - simplified circuit in terms of variable occurrence.




$$f = (a + b + c).(d + \bar{e}).x + y$$

Adapted from: Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019.

[3] Level gate implementation is a structured approach in digital circuit design where logic gates are organized into distinct levels. Each level processes signals from the previous one, and the complexity of the circuit is managed through this hierarchical organization. The first level consists of gates directly connected to the input variables of the circuit and perform initial operations on the inputs. Any levels between the first and last are intermediate levels and they process the outputs of the previous level. The last level, also known as final level, consists of gates that produce the final output of the circuit, taking inputs from the last intermediate level. Each level adds a stage of delay to the signal as it propagates through the circuit. Therefore, minimizing the number of levels can help reduce the overall delay and improve the performance of the circuit. Figure 2.1 shows the AND gates at the first level, and the OR gate as second and final level, thus a 2-level implementation. When digital circuits have more than two levels, they can also be referred to as "multilevel", like the example in Figure 2.2.

## 2.6 Relationship with Set Theory

Boolean algebra shares a deep connection with set theory, as both rely on similar logical operations such as union, intersection, and complementation. This relationship allows for the application of Boolean principles in a wide range of mathematical contexts, particularly when dealing with sets and their interactions. For an introductory exploration of the parallels between Boolean algebra and set theory, please refer to Appendix II.

## 2.7 Conclusion

In conclusion, Boolean algebra provides a powerful framework for analyzing and simplifying logical expressions, serving as a cornerstone in the design and function of digital circuits and computer systems. By mastering fundamental concepts, identities, and properties, one can efficiently manipulate and optimize logical operations. De Morgan's laws and techniques for algebraic simplification are particularly valuable for reducing complex expressions, enhancing the performance of digital systems.

This chapter lays the groundwork for understanding how abstract logic directly influences practical applications in computing. From its fundamental concepts to the laws and mathematical structures that define it, Boolean algebra emerges as a versatile and indispensable tool. With this knowledge in hand, it will be explored next how Boolean algebra finds tangible expression in the design and analysis of digital circuits.

# CHAPTER 3 : LOGIC GATES

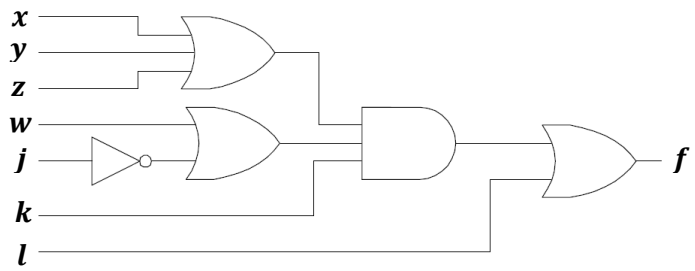
## 3.1 Introduction to Logic Gates in Digital Circuits

As introduced in the previous chapters, logic gates are known as the elemental building blocks of digital circuits, essential for processing and manipulating binary information. This chapter discusses the various types of logic gates and their practical implementation in digital circuits, which form the basis for more complex systems like combinational and sequential circuits. These gates perform operations that correspond to Boolean algebra, turning abstract logical expressions into real, functioning hardware components.

Digital circuits are hardware components that manipulate binary information. They process binary information, making decisions based on the state of input signals (0 or 1) to produce specific output signals.

Considered the foundation of modern computing, logic gates perform essential logical operations. These physical devices implement Boolean functions, simplifying circuit design by allowing designers to focus on the logical behavior of gates rather than their internal electronic details. Each gate performs a specific operation, and their outputs connect to the inputs of other gates to form complex digital circuits, as illustrated in Figure 3.1.

**Figure 3.1:** Example of a digital circuit.




As already discussed, associated with the binary variables in Boolean algebra are three basic logical operations: NOT, AND, and OR. Nonetheless, in addition to these three, there are more types of logic operations.

## 3.2 Logic Gates

Logic gates are used to create digital circuits that process binary information. These electronic circuits operate on one or more binary input signals to produce a binary output based on predefined logical rules.

Typically, voltage levels represent binary values, with a positive voltage indicating "1" and zero voltage indicating "0," though this can vary by circuit design (Mano & Kime, 2014). Gates respond to binary inputs within defined voltage ranges, with transitions occurring between these ranges. A key property of logic gates is gate delay, the time it takes for an input change to produce a corresponding output change. This delay accumulates in circuits with multiple gates, impacting overall system performance (Mano & Kime, 2014).

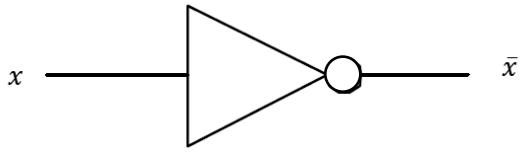
Understanding logic gates is crucial to the design and operation of computers and other digital systems, as they form the logical foundation for complex digital functions.

## 3.3 Types of Logic Gates

As previously discussed, this chapter will start with the three basic logic functions: NOT, AND, and "Inclusive-OR".

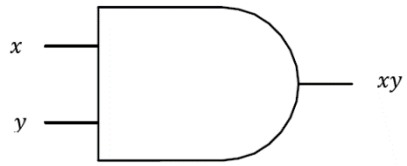
The NOT gate, commonly called the inverter, implements the NOT function and its representation is shown in Figure 3.2 below. Therefore, this function has a single input, $x$, and a single output: $x'$ or $\bar{x}$.

**Figure 3.2:** Example of a NOT gate.



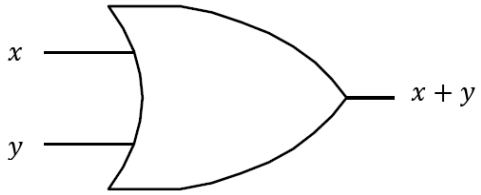
The AND gate implements the AND function and is represented in Figure 3.3 below. Hence the Boolean expression $x \cdot y$ or $xy$, represents the output term of a two input $(x, y)$ AND gate.

**Figure 3.3:** Example of a AND gate.



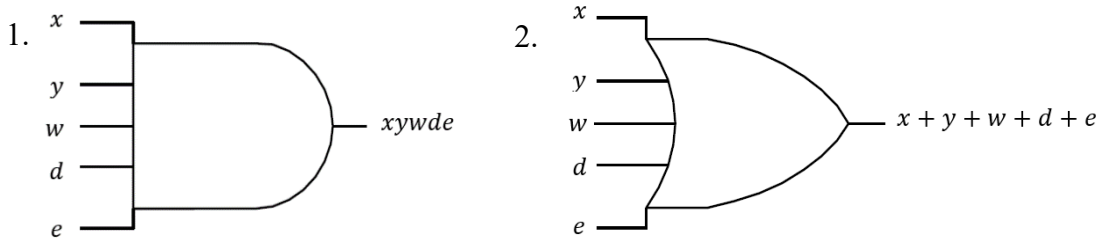

The OR function, or "inclusive-OR", is implemented by the OR gate, which is illustrated in Figure 3.4 below. Thus, the Boolean expression $x + y$, represents the output term of a two input $(x, y)$ OR gate.

**Figure 3.4:** Example of an OR gate.




These last two gates can have more than two inputs, as illustrated in Figure 3.5 below. These gates produce specific logical outputs based on the logic states of their multiple inputs.

**Figure 3.5:** Example of multiple input AND (1) and OR (2) gates.




There are sixteen functions of two Boolean variables, as shown in Table 3.1 below.

**Table 3.1:** Functions of two Boolean variables.

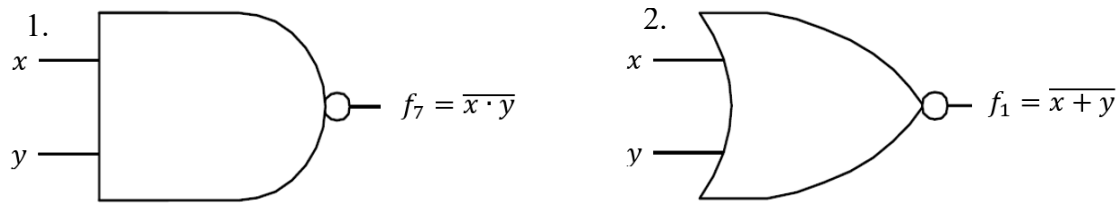
| $x$ | $y$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | $f_6$ | $f_7$ | $f_8$ | $f_9$ | $f_{10}$ | $f_{11}$ | $f_{12}$ | $f_{13}$ | $f_{14}$ | $f_{15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

The functions $f_8$ and $f_{14}$ represent the functions AND and OR respectively. Together with the NOT function, these are known as the basic Boolean functions, because one can write all other Boolean expressions with just these three functions alone. However, this could lead to very complex and inefficient circuits. For this reason, the use of other functions (and therefore logic gates) like NAND, NOR etc. is essential for practical, efficient, and optimized digital circuit design. These gates help simplify circuit implementation, improve performance, reduce costs, and enhance overall design flexibility.

The functions $f_1$ and $f_7$ represent the universal functions NOR and NAND respectively. In practical terms, the gates that implement these logic functions work just like an OR gate or an AND gate followed by a NOT gate for a NOR and NAND gate respectively.

As imagined for the placement of a NOT gate, these gates produce complementary outputs to OR and AND gates respectively. Figure 3.6 below illustrates these gates and respective functions. In fact, NAND and NOR logic gates are known as universal gates due to their inexpensive production costs and because "any Boolean function can be constructed using only NAND or only NOR gates" (Yang, 2020:8).

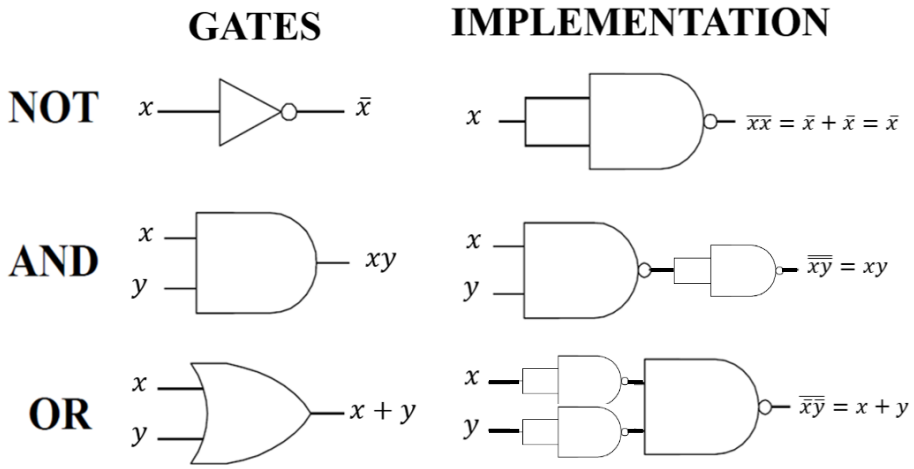
**Figure 3.6:** NAND (1) and NOR (2) gates and respective functions.



Both NAND and NOR gates can have more than two inputs, as seen in Figure 3.7 below. Because these gates are universal and more inexpensive, they are typically used in the world of digital circuits to implement other functions. Figure 3.8 below shows the implementation of NOT, AND and OR gates using NAND gates.

**Figure 3.7:** Multiple inputs NAND (1) and NOR (2) gates.

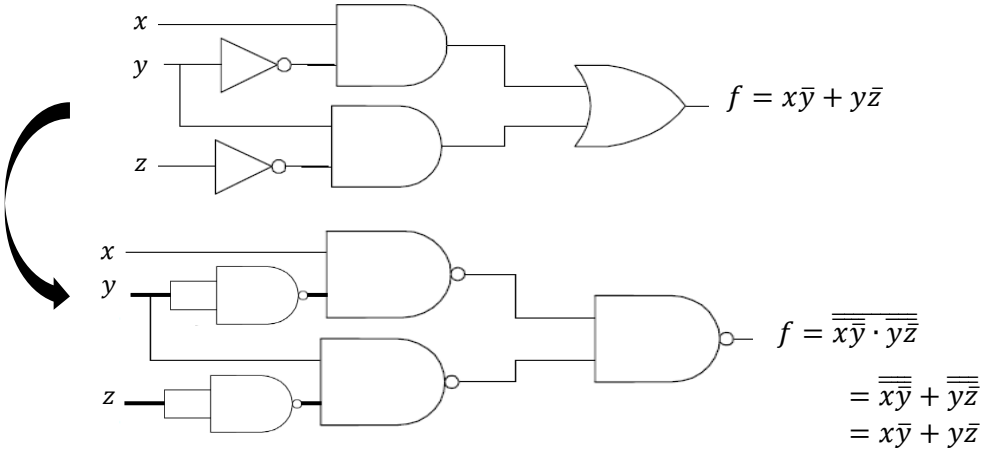

**Figure 3.8**: Implementation of NOT, AND and OR functions with NAND gates.




Adapted from: Mano & Kime, 2014.

Figure 3.9 below shows the transformation of a circuit, through the application of De Morgan's law, resulting in its implementation with only NAND gates, without changing the structure of the circuit (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

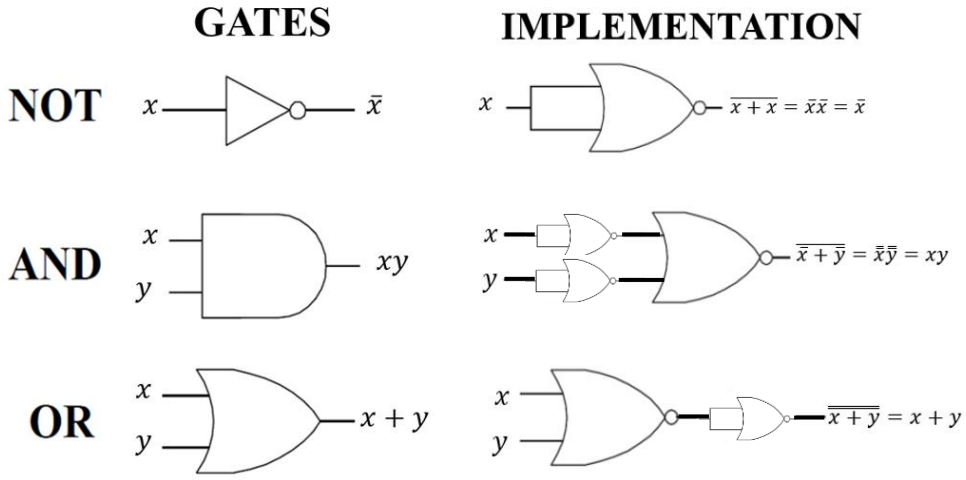
**Figure 3.9:** Digital circuits with NAND gates.




Adapted from: Mano & Kime, 2014.

Just like the NAND gate, the NOR gate is also universal, i.e., can represent any other function. Figure 3.10 below shows the implementation of NOT, AND and OR gates using NOR gates (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

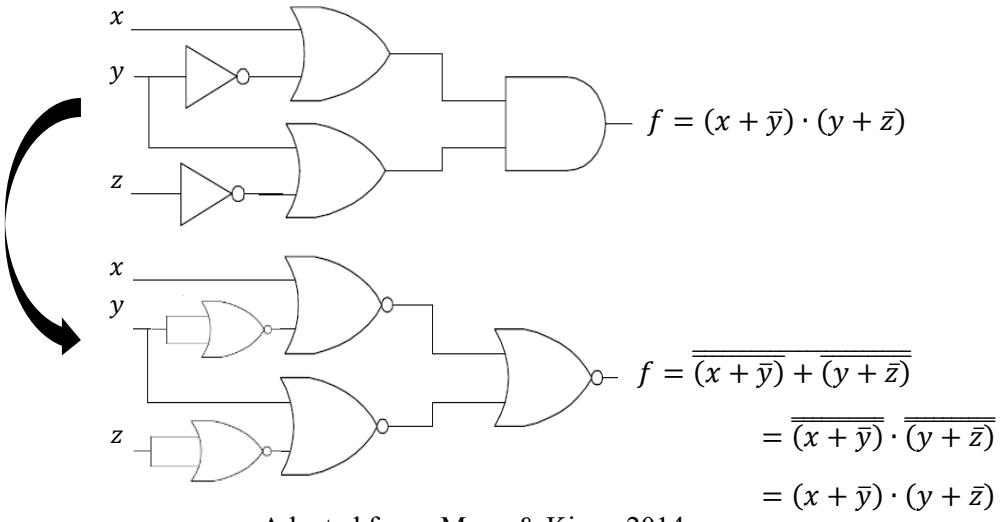
**Figure 3.10:** Implementation of NOT, AND and OR functions with NOR gates.




Adapted from: Mano & Kime, 2014.

Recovering the example given in Figure 3.9, a circuit can also be transformed, keeping its structure intact, using exclusively NOR gates, as seen in Figure 3.11 below (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

**Figure 3.11:** Digital circuits with NOR gates.

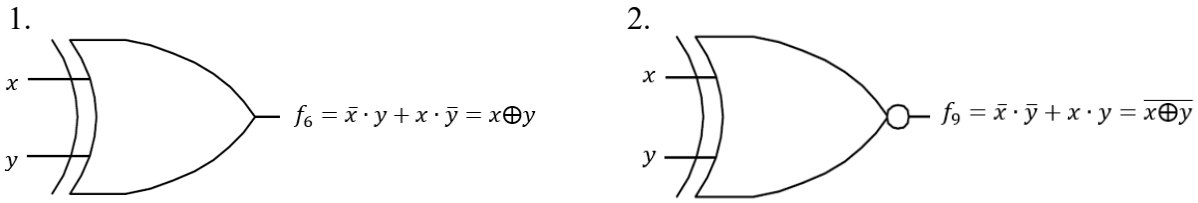



Adapted from: Mano & Kime, 2014.

Additionally, the functions $f_6$ and $f_9$ in Table 3.1 above, represent the logic functions XOR and XNOR respectively. For the logic gate XOR (or Exclusive-OR), the output will be TRUE only if one of the two outputs are TRUE, as seen in Table 3.1 above. This behavior makes the XOR gate particularly useful in various applications, including binary addition, error detection, and communications (Mano & Kime, 2014; Arroz, Monteiro & Oliveira, 2019).

The logic gate XNOR, as the name indicates, is the junction of the logic functions XOR and NOT. Therefore, its output will be complementary with the output of a XOR logic gate, as seen in Table 3.1 above (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). These gates and the correspondent logic functions are represented in Figure 3.12 below.

**Figure 3.12:** XOR (1) and XNOR (2) gates and respective functions.




Adapted from: Mano & Kime, 2014:61.

The logic gate XOR is also available with multiple inputs. In this case, the output will be TRUE only if the number of inputs that are TRUE are odd (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). This is usually helpful in error detection.

## 3.4 Conclusion

From the fundamental operations of AND, OR, and NOT gates to the more complex XOR and XNOR gates, these logical building blocks lay the groundwork for advanced computational systems.

The utilization of Boolean Algebra in the modeling of digital circuits forms the cornerstone of modern digital design, facilitating the development of advanced technologies that underpin our interconnected and technology-driven world. The next chapters will further explore the practical application of these gates in the modeling and design of digital circuits, guided by the principles of Boolean algebra.

# CHAPTER 4 : MODELLING OF DIGITAL CIRCUITS

## 4.1 Introduction to the Modelling of Digital Circuits

This chapter will explore techniques such as minterms, maxterms, sum of products, and product of sums for modeling complex circuits, as well as optimization techniques like Karnaugh maps (K-maps), which simplify Boolean expressions and minimize circuit complexity.

In the world of digital electronics, the efficient representation and analysis of complex circuits are essential for designing and optimizing electronic systems. As previously discussed, Boolean algebra serves as a powerful mathematical tool for modeling and simplifying digital circuits, providing a systematic way to express and manipulate binary logic. This methodological approach allows engineers and designers to translate real-world problems into logical expressions, enabling the creation of accurate and efficient digital circuits.

The design of these circuits involves translating logical requirements into a tangible circuit layout. Boolean expressions act as the blueprint, defining the relationship between inputs and outputs. The systematic application of logic gates enables the creation of circuits that execute specific functions.

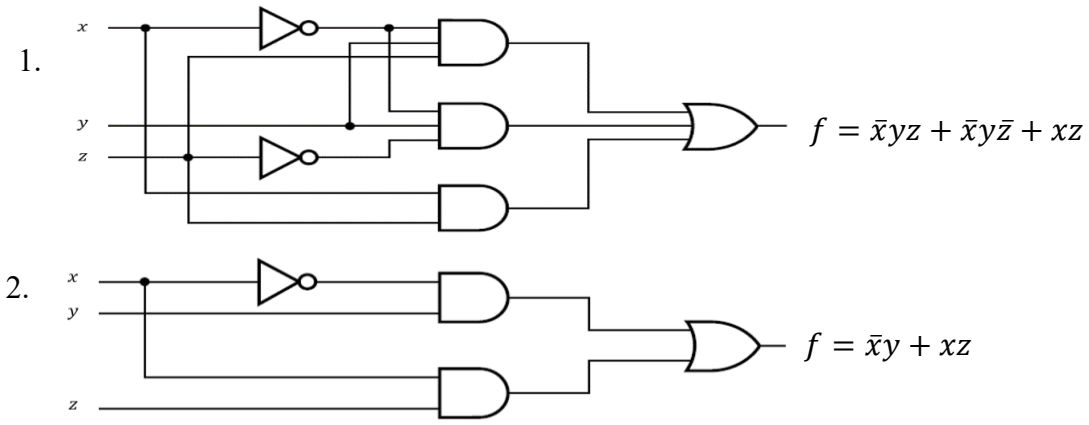
## 4.2 Minterms and Maxterms

In the world of Boolean algebra, a *variable* is a symbol $(x, y, z, \dots)$ that may take on the value of 0 or 1. A *literal* is a variable or its negation. Finally, a *term* is an expression formed by literals and operations at a certain level[4]. When implementing a Boolean expression using logic gates, each term (either a single literal or an operation on variables) corresponds to a distinct level in the circuit. For example, in Figure 4.1.1 below, $f = \bar{x}yz + \bar{x}y\bar{z} + xz$ has

[4] Resuming the explanation given in Chapter 2, level gate implementation refers to the arrangement of logic gates in a hierarchical structure where each layer or level of gates processes the outputs of the previous level and serves as inputs to the next level. Because each level adds a stage of delay to the signal, minimizing the number of levels improves the performance of the circuit.

three variables $x, y, z$; has the occurrence of eight literals $(\bar{x}, y, z, \bar{x}, y, \bar{z}, x, z)$ and three terms at the first level: $\bar{x}yz, \bar{x}y\bar{z}, xz$, and one term at the second level: obtained by the OR operation. On the other hand, in Figure 4.1.2, $f = \bar{x}y + xz$ has the same variables, but only has the occurrence of four literals $(\bar{x}, y, x, z)$. Reducing the number of terms, literals, or both, in a Boolean expression allows for the simplification of the resulting circuit and makes it more inexpensive to implement.

**Figure 4.1:** Representation of a more complex function 1. and its simpler version 2.




Source: Mano & Kime, 2014:47.

**Table 4.1:** Truth table for the functions in Figure 4.1.

| $x$ | $y$ | $z$ | 1. $f$ | 2. $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Source: Mano & Kime, 2014:47.

As seen in Table 4.1 above, both functions yield the same truth table, therefore they are equivalent. Consequently, both circuits produce the same output for every possible combination of the three input variables. While each circuit serves the same purpose, the one with fewer gates is preferable due to its efficiency, i.e., it requires fewer components to achieve the same result (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

Boolean algebra is used to reduce expressions to obtain simpler circuits. However, for highly complex functions, determining the best expression based solely on term and literal counts can be challenging, even with the help of computer programs (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). Nonetheless, computer tools for synthesizing logic circuits typically include methods for expression reduction, which can offer good solutions if not necessarily the optimal ones.

To effectively simplify Boolean expressions, it's crucial to understand the fundamental building blocks of these expressions. This is where the concepts of minterms and maxterms come into play. Minterms and maxterms are essential tools in Boolean algebra that provide a systematic way to represent and manipulate logic functions. By breaking down complex functions into these basic components, it becomes easier to apply reduction techniques and achieve more efficient circuit designs.

A *minterm* is a product (AND operation) of all the variables in a Boolean function, each appearing exactly once in either its TRUE form (*uncomplemented*) or its FALSE form (*complemented*). For a Boolean function with $n$ variable, each row of the truth table represents a possible combination of these variables' values (0s and 1s) and there are $2^n$ such combinations for $n$ variables.

For example, Table 4.2 below shows that the function $f$ has three variables, $x, y$ and $z$, and in its truth table shows that this function has a total of $2^3 = 8$ possible combinations $(000, 001, 010, 011, 100, 101, 110, 111)$, and thus a total of eight minterms.

Conventionally, each minterm is denoted as $m_i$ where the index $i$ indicates the decimal number equivalent to the binary combination it represents[5], like illustrated in Table 4.2 below.

To write the minterms for a specific row of the truth table, each variable is included and if it has a value of 1 in that row, the variable itself is used; if its value in that row is 0, its complemented form (or negation) will be used instead.

For example, in the first row of Table 4.2 below, the variables $x, y$ and $z$, all have a value 0, therefore the minterm for this row is written with the complements of these variables $\bar{x}\bar{y}\bar{z}$. However, in the second row of the table, the variables $x$ and $y$ have a value of 0, while $z$ takes the value 1, thus the minterm for this row is given as $\bar{x}\bar{y}z$.

**Table 4.2:** Minterms for three variables.

| $x$ | $y$ | $z$ | ***Minterms*** | | $f$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | $\bar{x}\bar{y}\bar{z}$ | $m_0$ | 0 |
| 0 | 0 | 1 | $\bar{x}\bar{y}z$ | $m_1$ | 0 |
| 0 | 1 | 0 | $\bar{x}y\bar{z}$ | $m_2$ | 0 |
| 0 | 1 | 1 | $\bar{x}yz$ | $m_3$ | 1 |
| 1 | 0 | 0 | $x\bar{y}\bar{z}$ | $m_4$ | 0 |
| 1 | 0 | 1 | $x\bar{y}z$ | $m_5$ | 0 |
| 1 | 1 | 0 | $xy\bar{z}$ | $m_6$ | 0 |
| 1 | 1 | 1 | $xyz$ | $m_7$ | 0 |

Each Boolean function can be expressed as a sum of minterms for which the function's output is 1, which is called the *sum of products* form. These minterms are of particular interest, because they define the conditions under which the function evaluates to TRUE ($f$ =

[5] Referring to the binary system discussed in Chapter 1, $m_0$ corresponds to the binary combination 000, which is 0 in decimal; $m_1$ corresponds to 001, which is 1 in decimal, and so on. Each combination is listed in the table according to its binary representation, while the index $i$ of each minterm (or maxterm) indicates its equivalent decimal value. This systematic approach allows for an organized representation of all possible variable combinations in both binary and decimal forms.

1). Thus, by focusing on these minterms, a simplified and efficient representation of the Boolean function[6] can be constructed.

The sum of products form, also known as disjunctive normal form, is a standardized way of representing Boolean functions, where the function is written as the sum (OR operation) of its true minterms. This form is useful for designing digital circuits because it clearly shows the conditions under which the function outputs 1. By focusing on the minterms for which the function is 1, techniques like Karnaugh maps can be used to simplify the Boolean expression. As said before, simplifying the expression reduces the number of gates and inputs required in the actual digital circuit, leading to more efficient designs. This will be explained in more depth in the following sections.

*Maxterms* are the dual concept to minterms. A maxterm is a sum (OR operation) of all the variables in the function, each appearing exactly once, either in its TRUE form (*uncomplemented*) or its FALSE form (*complemented*). As it is true for minterms, a Boolean function with $n$ variables, also has $2^n$ maxterms.

For example, Table 4.3 below shows that the function $f$ has three variables, $x, y$ and $z$, and in its truth table shows that this function has a total of $2^3 = 8$ possible combinations $(000, 001, 010, 011, 100, 101, 110, 111)$, and thus a total of eight maxterms.

Conventionally, each maxterm is denoted as $M_i$, where $i$ indicates the decimal number equivalent to the binary combination it represents. Table 4.3 below also illustrates an example of maxterms for three variables.

To write the maxterms for a specific row of the truth table, each variable is included and if it has a value of 0 in that row, the variable itself will be used; if its value in that row is 1, its complemented form (or negation) will be used instead.

[6] In this initial example, there is only one minterm that makes $f = 1$; hence, $f$ would be expressed as $\bar{x}yz$. More complex examples featuring explicit sums of minterms will be presented later.

For example, in the first row of Table 4.3 below, the variables $x, y$ and $z$, all have a value 0, therefore the maxterm is $x + y + z$. On the other hand, in the second row of the table, the variables $x$ and $y$ have a value of 0, while $z$ takes the value 1, thus the maxterm for this row is $x + y + \bar{z}$.

**Table 4.3:** Maxterms for three variables.

| $x$ | $y$ | $z$ | ***Maxterms*** | | $f$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | $x + y + z$ | $M_0$ | 1 |
| 0 | 0 | 1 | $x + y + \bar{z}$ | $M_1$ | 1 |
| 0 | 1 | 0 | $x + \bar{y} + z$ | $M_2$ | 1 |
| 0 | 1 | 1 | $x + \bar{y} + \bar{z}$ | $M_3$ | 0 |
| 1 | 0 | 0 | $\bar{x} + y + z$ | $M_4$ | 1 |
| 1 | 0 | 1 | $\bar{x} + y + \bar{z}$ | $M_5$ | 1 |
| 1 | 1 | 0 | $\bar{x} + \bar{y} + z$ | $M_6$ | 1 |
| 1 | 1 | 1 | $\bar{x} + \bar{y} + \bar{z}$ | $M_7$ | 1 |

Each Boolean function can be expressed as a product of maxterms for which the function's output is 0, which is called the *product of sums* form. These maxterms are of particular interest, because they define the conditions under which the function evaluates to FALSE ($f = 0$). Thus, by focusing on these maxterms, a simplified and efficient representation of the Boolean function[7] can be created.

The product of sums form, also known as conjunctive normal form, is a standardized way of representing Boolean functions, where the function is written as the product (AND operation) of its false maxterms. This form is useful for designing digital circuits because it clearly shows the conditions under which the function outputs 0. By focusing on the maxterms for which the function is 0, techniques like Karnaugh maps can be used to simplify the

[7] In this initial example, there is only one maxterm that makes $f = 0$; hence, $f$ would be expressed as $x + \bar{y} + \bar{z}$. More complex examples featuring explicit products of maxterms will be presented later.

Boolean expression. Thus, reducing the number of gates and inputs required in the actual digital circuit and leading to more efficient designs. This will be explained in more depth in the following sections.

In Boolean algebra, minterms and maxterms with the same index $i$ are complements of each other such as $m_i = \overline{M_i}$, mirroring the complementary nature of the basic Boolean operations AND and OR.

The AND operation (product) and OR operation (sum) are basic Boolean operations that exhibit complementarity. In the context of minterms and maxterms, AND is used to combine variables within a minterm, making it TRUE for a specific combination of inputs. While OR is used to combine variables within a maxterm, making it FALSE for a specific combination of inputs.

As previously explained, a Boolean function can be expressed as the sum of minterms for which the function is TRUE (sum of products form), and the same function can be expressed as the product of maxterms for which the function is FALSE (product of sums form). These forms are duals of each other. As illustrated in Chapter 2, the duality principle in Boolean algebra states that every Boolean expression remains valid if AND and OR operations are interchanged (or 0s and 1s).

The complementarity of AND and OR can be demonstrated by De Morgan's laws such that $\overline{xy} = \bar{x} + \bar{y}$ and $\overline{x + y} = \bar{x}\bar{y}$. These laws show how the negation of an AND operation results in an OR operation and vice versa.

Minterms and maxterms cover all possible combinations of variables, and together, they exhaustively represent the function. In Table 4.4 below, the minterm $m_1 = \bar{x}\bar{y}z$ is TRUE for $x = 0, y = 0, z = 1$; while the maxterm $M_1 = x + y + \bar{z}$ is FALSE for $x = 0, y = 0, z = 1$. In (4.1) below it is proven that $m_1 = \overline{M_1}$.

**Table 4.4:** Complementary of minterms and maxterms.

| $x$ | $y$ | $z$ | ***Minterms*** | | ***Maxterms*** | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | $\bar{x}\bar{y}\bar{z}$ | $m_0$ | $x+y+z$ | $M_0$ |
| 0 | 0 | 1 | $\bar{x}\bar{y}z$ | $\boldsymbol{m_1}$ | $x+y+\bar{z}$ | $\boldsymbol{M_1}$ |
| 0 | 1 | 0 | $\bar{x}y\bar{z}$ | $m_2$ | $x+\bar{y}+z$ | $M_2$ |
| 0 | 1 | 1 | $\bar{x}yz$ | $m_3$ | $x+\bar{y}+\bar{z}$ | $M_3$ |
| 1 | 0 | 0 | $x\bar{y}\bar{z}$ | $m_4$ | $\bar{x}+y+z$ | $M_4$ |
| 1 | 0 | 1 | $x\bar{y}z$ | $m_5$ | $\bar{x}+y+\bar{z}$ | $M_5$ |
| 1 | 1 | 0 | $xy\bar{z}$ | $m_6$ | $\bar{x}+\bar{y}+z$ | $M_6$ |
| 1 | 1 | 1 | $xyz$ | $m_7$ | $\bar{x}+\bar{y}+\bar{z}$ | $M_7$ |

$$\boldsymbol{m_1} = \bar{x}\bar{y}z = \overline{x+y+\bar{z}} = \overline{\boldsymbol{M_1}} \tag{4.1}$$

### 4.3 Sum of Products

The sum of minterms form is, as previously mentioned, a common algebraic expression which can be derived directly from a truth table.

This sum includes all minterms for which the function evaluates to 1, i.e., each selected minterm represents a specific combination of input variables that results in a TRUE output for the function. The sum of these minterms expression represents the Boolean function as a logical sum (OR) of its individual minterms and is particularly useful for implementing Boolean functions using logic gates.

Taking the truth table in Table 4.5 below, the Boolean function $xy + \bar{z}$ can be represented through a sum of products.

**Table 4.5:** Truth table for the Boolean function $xy + \bar{z}$.

| $x$ | $y$ | $z$ | $\bar{z}$ | $xy$ | $xy + \bar{z}$ | ***Minterms*** | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | $\bar{x}\bar{y}\bar{z}$ | $m_0$ |
| 0 | 0 | 1 | 0 | 0 | 0 | $\bar{x}\bar{y}z$ | $m_1$ |
| 0 | 1 | 0 | 1 | 0 | 1 | $\bar{x}y\bar{z}$ | $m_2$ |
| 0 | 1 | 1 | 0 | 0 | 0 | $\bar{x}yz$ | $m_3$ |
| 1 | 0 | 0 | 1 | 0 | 1 | $x\bar{y}\bar{z}$ | $m_4$ |
| 1 | 0 | 1 | 0 | 0 | 0 | $x\bar{y}z$ | $m_5$ |
| 1 | 1 | 0 | 1 | 1 | 1 | $xy\bar{z}$ | $m_6$ |
| 1 | 1 | 1 | 0 | 1 | 1 | $xyz$ | $m_7$ |

The function $f$ is TRUE when the variable combinations are the ones on lines 0, 2, 4, 6 and 7. Thus, $f$ is TRUE when the combinations given by $m_0$ *or* $m_2$ *or* $m_4$ *or* $m_6$ *or* $m_7$*:* $xy + \bar{z} = \sum_{i \in P} m_i = \bar{x}\bar{y}\bar{z} + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z} + xyz,\ P = \{0,2,4,6,7\}$.

The sum of products form includes the maximum number of literals in each term and often has more product terms than needed. This happens because, by definition, each minterm must encompass all variables of the function, whether complemented or uncomplemented.

The sum of minterms is derived from the truth table. Then, begins the process of simplifying the expression in order to minimize the number of product terms and the number of literals in each term and, therefore, simplifying the circuit to be implemented.

## 4.4 Product of Sums

In a similar way to that of minterms, maxterms are also used to represent Boolean functions. The product of maxterms form is an algebraic expression known as the product of sums and can be directly derived from a given truth table.

This product includes all maxterms for which the function evaluates to 0, i.e., each selected maxterm represents a specific combination of input variables that results in a FALSE output for the function. The product of these maxterms expression represents the Boolean function as a logical product (AND) of its individual maxterms. Recovering the concept of complementarity between minterms and maxterms, Table 4.5 above allows for the deduction the maxterms of this Boolean function as shown in Table 4.6 below.

**Table 4.6:** Truth table for the Boolean function $xy + \bar{z}$ with maxterms.

| $x$ | $y$ | $z$ | $\bar{z}$ | $xy$ | $xy + \bar{z}$ | ***Minterms*** | | | ***Maxterms*** | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | $\bar{x}\bar{y}\bar{z}$ | $\boldsymbol{m_0}$ | | $x + y + z$ | $M_0$ |
| 0 | 0 | 1 | 0 | 0 | 0 | $\bar{x}\bar{y}z$ | $m_1$ | | $x + y + \bar{z}$ | $\boldsymbol{M_1}$ |
| 0 | 1 | 0 | 1 | 0 | 1 | $\bar{x}y\bar{z}$ | $\boldsymbol{m_2}$ | | $x + \bar{y} + z$ | $M_2$ |
| 0 | 1 | 1 | 0 | 0 | 0 | $\bar{x}yz$ | $m_3$ | $\boldsymbol{m = \bar{M}}$ | $x + \bar{y} + \bar{z}$ | $\boldsymbol{M_3}$ |
| 1 | 0 | 0 | 1 | 0 | 1 | $x\bar{y}\bar{z}$ | $\boldsymbol{m_4}$ | | $\bar{x} + y + z$ | $M_4$ |
| 1 | 0 | 1 | 0 | 0 | 0 | $x\bar{y}z$ | $m_5$ | | $\bar{x} + y + \bar{z}$ | $\boldsymbol{M_5}$ |
| 1 | 1 | 0 | 1 | 1 | 1 | $xy\bar{z}$ | $\boldsymbol{m_6}$ | | $\bar{x} + \bar{y} + z$ | $M_6$ |
| 1 | 1 | 1 | 0 | 1 | 1 | $xyz$ | $\boldsymbol{m_7}$ | | $\bar{x} + \bar{y} + \bar{z}$ | $M_7$ |

The idea behind the construction of the product of sums is to look at the combinations for which $f$ is FALSE. In this way, $f$ is TRUE when it is not FALSE, i.e., $f$ is true when the combination of lines 1, 3 and 5 don't verify: $xy + \bar{z} = \prod_{i \in P} M_i = (x + y + \bar{z})(x + \bar{y} + \bar{z})(\bar{x} + y + \bar{z}),\ P = \{1,3,5\}$.

The product of sums uses the maxterms corresponding to the minterms not used in the sum of products, i.e., it's equivalent to applying De Morgan's law to the complement of the function. Figure 4.2 below illustrates the conversion between these two canonical forms.

**Figure 4.2:** Conversion between sum of products and product of sums.

$$\boldsymbol{f = m_0 + m_1 + m_3 + m_5 + m_7}$$

$$\bar{f} = \boldsymbol{m_2 + m_4 + m_6}$$

$$\leftrightarrow f = \overline{\bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z}}$$

$$\leftrightarrow f = \overline{\bar{x}y\bar{z}}\ \overline{x\bar{y}\bar{z}}\ \overline{xy\bar{z}}$$

$$\leftrightarrow f = (x + \bar{y} + z)(\bar{x} + y + z)(\bar{x} + \bar{y} + z)$$

$$\leftrightarrow f = \boldsymbol{M_2 M_4 M_6}$$

| $x$ | $y$ | $z$ | $f$ | $m_i$ | $M_i$ | | $\bar{f}$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | $\boldsymbol{m_0}$ | $M_0$ | | 0 |
| 0 | 0 | 1 | 1 | $\boldsymbol{m_1}$ | $M_1$ | | 0 |
| 0 | 1 | 0 | 0 | $m_2$ | $\boldsymbol{M_2}$ | | 1 |
| 0 | 1 | 1 | 1 | $\boldsymbol{m_3}$ | $M_3$ | | 0 |
| 1 | 0 | 0 | 0 | $m_4$ | $\boldsymbol{M_4}$ | | 1 |
| 1 | 0 | 1 | 1 | $\boldsymbol{m_5}$ | $M_5$ | | 0 |
| 1 | 1 | 0 | 0 | $m_6$ | $\boldsymbol{M_6}$ | | 1 |
| 1 | 1 | 1 | 1 | $\boldsymbol{m_7}$ | $M_7$ | | 0 |

$$\boldsymbol{M_2 M_4 M_6} = f = \bar{\bar{f}} = \overline{\boldsymbol{m_2 + m_4 + m_6}}$$

## 4.5 Karnaugh Maps

The level of complexity involved in the implementation of a Boolean function through digital logic gates directly correlates with the complexity of the algebraic expression itself. While the truth table representation of a function is unique, its algebraic expression can take several equivalent forms (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

The algebraic simplification process of Boolean expressions was already discussed, nevertheless, this process lacks predictable rules for each step. Another resource employed to simplify Boolean expressions are *Karnaugh Maps* (or K-maps). This is a systematic approach, whose primary goal is to minimize the number of logic gates in a circuit, which, in turn, enhances the circuit's speed, reduces power consumption, and improves overall efficiency.

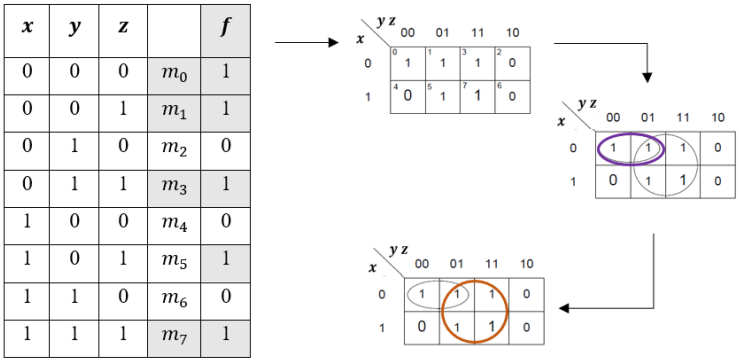
Contrary to truth tables, that represent relationships between logic inputs and desired outputs in a tabular form, a K-map is organized as a grid, visually representing all possible expressions of a function in standard form using squares, each representing a minterm.

As previously said, any Boolean function can be represented as a sum of minterms. In this way, the K-map identifies the function graphically by enclosed squares containing the included minterms. Because of its structure, the K-map offers a visual representation of the function's different expressions, allowing for the identification of patterns and derivation of alternative algebraic expressions. The resulting simplified expressions from the map are always in either sum-of-products or product-of-sums forms.

Through this process, it is assumed that "the simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term" (Mano & Ciletti, 2013:74). This expression yields a circuit with minimal gates and inputs per gate. However, it's possible to find multiple expressions meeting these criteria, and either solution suffices.

While the inputs on a truth table commonly follow a standard binary sequence as, for example, 00, 01, 10, 11 for two variables; the input values on a K-map must be organized so that the values for adjacent[8] columns (and rows) differ by only a single bit. This is known as Gray code ordering. For example, for two variables, the column headings would be arranged as 00, 01, 11, and 10, instead of the standard binary sequence. This arrangement ensures that any two adjacent cells in the K-map differ by only one bit, facilitating the identification and grouping of adjacent ones to simplify Boolean expressions. This process is illustrated in Figure 4.3 below.

**Figure 4.3:** Karnaugh Maps.



| $x$ | $y$ | $z$ | | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | $m_0$ | 1 |
| 0 | 0 | 1 | $m_1$ | 1 |
| 0 | 1 | 0 | $m_2$ | 0 |
| 0 | 1 | 1 | $m_3$ | 1 |
| 1 | 0 | 0 | $m_4$ | 0 |
| 1 | 0 | 1 | $m_5$ | 1 |
| 1 | 1 | 0 | $m_6$ | 0 |
| 1 | 1 | 1 | $m_7$ | 1 |

[8] Two squares are said to be adjacent in logical terms when only one logical variable changes its value in the representation of these squares.

Following the example given above in Figure 4.3, it is circled in purple that the term is 1 when $x = 0$; and $y = 0$; and ($z = 0$ or $z = 1$). This can be written in a simplified form as $\bar{x}\bar{y}(\bar{z} + z) = \bar{x}\bar{y}$ (Mano & Kime, 2014). Similarly, it is circled in orange that the term is 1 when ($x = 0$ or $x = 1$) and ($y = 0$ or $y = 1$) and $z = 1$. This can be simplified as $(\bar{x} + x)(\bar{y} + y)z = z$ (Mano & Kime, 2014).

## 4.6 Karnaugh's Minimization

The process of creating a K-map starts with truth table conversion, i.e., creating a truth table for the given Boolean expression, listing all possible input combinations and their corresponding output values, and then mapping inputs, i.e., transfer the information from the truth table to the K-map. This was shown in Figure 4.3 above.

Then, the map is examined for adjacent 1s and the isolated 1s are grouped[9] (or *looped*). After this, the map is studied for 1s that are only adjacent to one other 1 and the pair in this condition is looped. Then, any octet[10] is looped even if the 1s in it have already been looped. The following step consists of looping any quad[11] containing one or more 1s that have not been previously looped, shown in Figure 4.3 above (*orange circle*). It is important in this step to use the minimum number of loops.

Following this rule, any pairs required to include 1s which have not yet been looped should be looped, and, at the end of this process, each group should include 1, 2, 4, 8, or any

[9] In addition to adjacent cells horizontally and vertically, K-maps also allow for "**wrap-around" adjacency**, meaning that the cells on the edges and corners of the map are considered adjacent to the cells on the opposite edge or corner.

[10] In Karnaugh maps, an **octet** refers to a group of eight adjacent cells. Octets can be used to identify patterns in the truth table of a Boolean function and simplify the expression accordingly (Mano & Ciletti, 2013; Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). By combining adjacent cells in groups of eight, it becomes possible to optimize and reduce the logic expressions further, leading to more efficient implementations of digital circuits.

[11] In Karnaugh maps, a **quad** refers to a group of four adjacent cells. These cells can be combined or simplified together when minimizing Boolean functions (Mano & Ciletti, 2013; Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). By identifying and grouping adjacent cells in the Karnaugh map, it becomes easier to simplify and optimize the logic expressions.

power of 2 cells. For each group, it is created a simplified term based on the variables that do not change within the group and the combination of these terms forms the simplified Boolean expression (Mano & Kime, 2014).

An example that illustrates Karnaugh's minimization process in greater detail is given in Appendix III.

Proper grouping in K-maps helps to simplify the Boolean expression efficiently. The solution is always a product of sums or a sum of products, providing a valid logic design with the minimum number of terms and with the minimum number of inputs to each gate, which can translate in a favorable cost-effective solution (Mano & Ciletti, 2013).

The process of grouping in K-maps must follow a set of rules for effective grouping:

- Groupings can contain only 1s.
- Only 1s in adjacent cells can be grouped and diagonal grouping is not allowed.
- The number of 1s in a group must be a power of 2.
- Looping a pair (or quad, or octet etc.) of 1s eliminates the two variables that appear in both complemented and uncomplemented form.
- All 1s must belong to a group, even if it is a group of one.
- Overlapping groups are allowed and sometimes necessary to achieve the simplest form.
- Cells on the far edges or corners can be considered adjacent, i.e., wrap around is allowed.
- Use the fewest number of groups possible, i.e., each group should be as large as possible.

## 4.7 Conclusion

In conclusion, the modeling of digital circuits plays a crucial role in optimizing circuit design by representing complex logic functions in a simplified and efficient manner. Minterms and maxterms form the foundation for expressing Boolean functions, while techniques like the sum of products and product of sums provide systematic methods for structuring these expressions. Karnaugh maps offer a visual tool for minimizing Boolean functions, leading to more streamlined and effective digital circuit designs. By mastering these methods, designers can enhance the performance and efficiency of digital systems, ensuring more reliable and optimized circuit implementations.

The subsequent chapters will build upon this foundation, exploring two types of digital circuits and arriving at the world of machine learning, more precisely of neural networks.

# CHAPTER 5 : COMBINATIONAL CIRCUITS

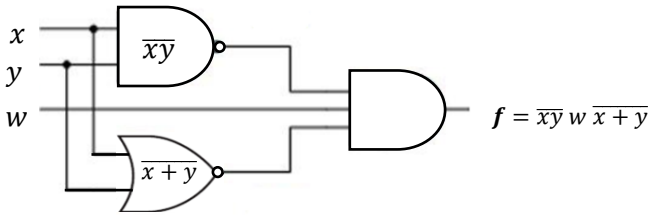
## 5.1 Introduction to Combinational Circuits

In this chapter, combinational circuits, such as adders, subtractors, decoders, and encoders, are explored and the practical application of the theory previously presented is demonstrated, showing how digital systems can be designed to perform specific tasks like arithmetic and data encoding.

Combinational circuits are digital circuits where the output depends solely on the current input values, without reference to past inputs or outputs. These stateless circuits generate deterministic outputs based on their logic design and are used in arithmetic operations, data processing, and control logic. All previously given examples were of combinational circuits.

Furthermore, these circuits may have multiple outputs, each representing a distinct Boolean function derived directly from the inputs. Thus, their behavior is governed by Boolean logic and can be represented by a logic diagram and truth table, though a single truth table can be implemented by different circuits, as illustrated in Figure 5.1 below.

**Figure 5.1:** Combinational circuit.




| $x$ | $y$ | $w$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

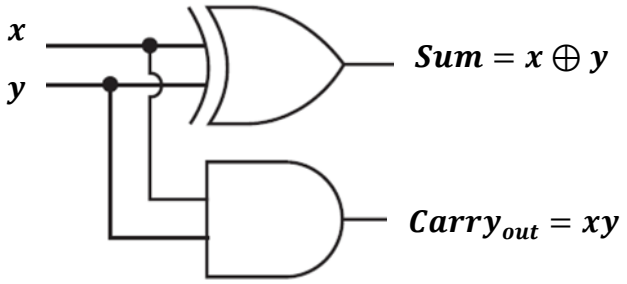
## 5.2 Adders and Subtractors

The combinational circuits that perform sums of bits are known as adders and can be found, for example, in the arithmetic logic unit (ALU) of modern computers. There are two types of adders: the half adder (HA) and the full adder (FA).

A HA is a simple digital circuit built from two logic gates and is used to add together two bits and produce two outputs: the sum of the digits ($S$) and a carry-out bit ($Carry_{out}$), being composed of two logic gates: a XOR gate and an AND gate. Figure 5.2 below shows an example of a logic diagram of a HA and its respective truth table.

The XOR gate takes two input bits, $x$ and $y$, and outputs 1 if exactly one of the input bits is 1, and 0 otherwise. This represents the $S$ bit of the addition. On the other hand, the AND gate takes two input bits, $x$ and $y$, and outputs 1 only if both input bits are 1. This represents the $Carry_{out}$ bit of the addition.

**Figure 5.2:** Half adder and its truth table.




| $x$ | $y$ | Sum | Carry-out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

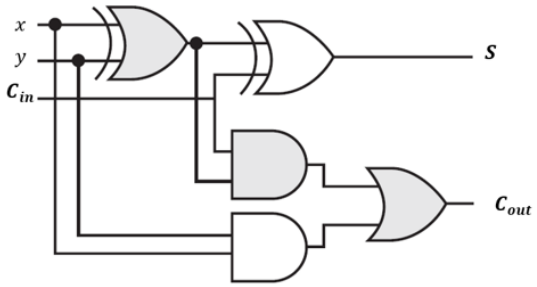
However, the HA can only start the operation, but because it misses the carry-in input, it is not able to perform the entire operation on its own. On the other hand, a FA takes three binary inputs: two bits to be added ($x$ and $y$) and a carry-in input ($C_{in}$) from a previous stage.

The FA computes the $S$ of the inputs and generates a $C_{out}$ output to propagate to the next stage of addition. Figure 5.3[12] below shows an example of a logic diagram of a FA and its respective truth table, as well as the K-map and how the Boolean expressions are calculated.

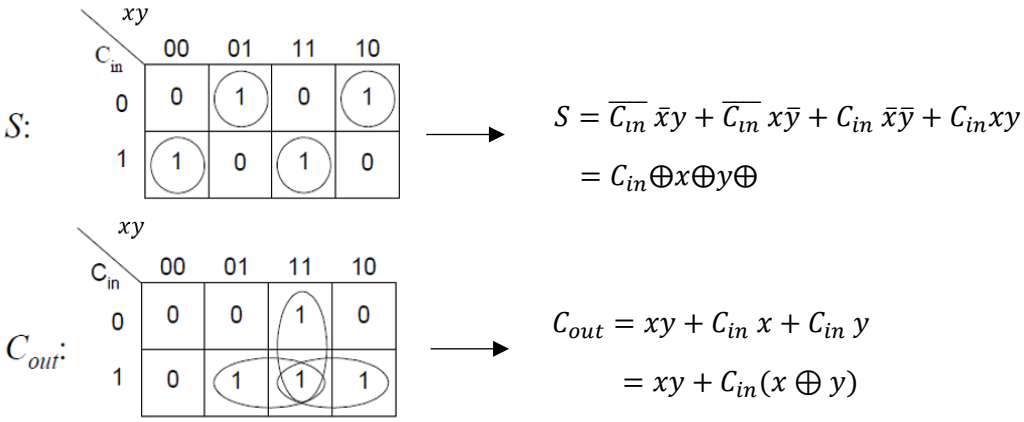
[12] The equations for the $S$ and $C_{out}$ of the FA presented in Figure 5.3 are proven in Appendix IV.

The bit of the $S$ outputs 1 only when one input or all three inputs are 1. On the other hand, the $C_{out}$ takes the value of 1 if two or three inputs are also 1.

**Figure 5.3:** Full-adder and its truth table.




| $x$ | $y$ | $C_{in}$ | S | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |




Adapted from: Bhagat; Mano & Kime, 2014.

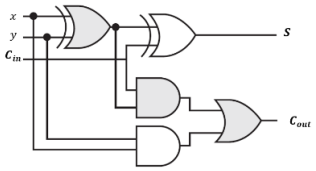
In reality, binary addition is performed by a circuit composed only of FAs, or by one HA connected to FAs, and has as many of these as bits to sum. So, taking the example in Figure 5.4 below, a sum of two numbers is composed of four bits each ($0111_2 = 7_{10}$) and ($0010_2 = 2_{10}$). To start the first stage of addiction, either an HA or a FA with $Carry_{in} = 0$ can be taken, because this operation starts without a $Carry$ and the $Carry_{out}$ of one stage will be the $Carry_{in}$ bit of the next one.

**Figure 5.4:** Addition of two numbers of four bits.

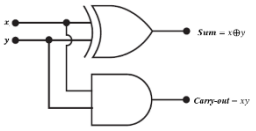
| | | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 1 | 0 | ← ***Carry.*** |
| 7 | | 0 | 1 | 1 | 1 ← Addend. |
| + 2 | + | 0 | 0 | 1 | 0 ← Addend. |
| 9 | | 1 | 0 | 0 | 1 ← Result. |

**A.** We start the operation without a *Carry* or with $Carry_{in} = 0$. For this reason, our circuit can begin with a HA or with a FA with $B_{in} = 0$.

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | |
| | 0 | 1 | 1 | 1 |
| + | 0 | 0 | 1 | 0 |
| | 1 | 0 | 0 | 1 |




or




| $x$ | $y$ | $C_{in}$ | S | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**B.** The operations will proceed according to the respective truth table.
**HA**: $S = 1 + 0 = 1,\ C_{out} = 0.$
**FA**: $S = 1 + 0, C_{in} = 0 \rightarrow S = 1, C_{out} = 0.$

| $x$ | $y$ | Sum | Carry-out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

| $x$ | $y$ | $C_{in}$ | S | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | |
| | 0 | 1 | 1 | 1 |
| + | 0 | 0 | 1 | 0 |
| | 1 | 0 | 0 | 1 |

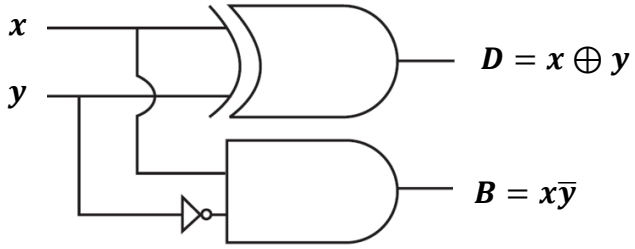
**C.** To perform the next stage in our sum, we have a FA with a $C_{in} = 0$ (from our previous stage), and we perform $1 + 1, C_{in} = 0$. According to its truth table, the **FA** outputs: $S = 0$ and $C_{out} = 1$.

**D.** And the process continues until the operation is complete.

Just like addition, subtraction with combinational circuits is also possible. Binary subtraction using 2's complement is a convenient method, frequently used, that simplifies the subtraction process by converting it into addition. A detailed explanation of this method can be found in Appendix V.

In the world of digital circuits, however, a circuit that will perform binary subtractions directly can be designed. These circuits are known as full subtractors (FS). An important integrant part of these are half subtractors (HS) that are illustrated in Figure 5.5 below and take two inputs the minuend, $x$, and the subtrahend, $y$, producing two outputs: the difference, $D$, and the borrow, $B$.

**Figure 5.5:** Half subtractor and its truth table.

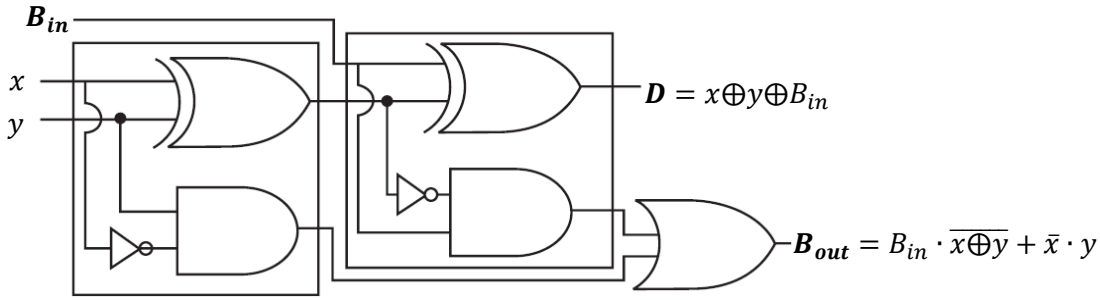



| $x$ | $y$ | D | B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

In Figure 5.5 above, the HS takes the subtrahend, $y$, and inverts it with a NOT gate, ANDing the result with the minuend, $x$, which yields the borrow. In parallel, a XOR gate takes the minuend, $x$, and the subtractor, $y$, outputting the difference. Depending on the values of our inputs, our circuit will output a difference and a borrow of 0 or 1, according to its truth table.

However, more than one HS is needed to perform binary subtractions. As it can been seen in Figure 5.6 below, similar to addition with logic gates, a FS can be obtained by integrating two HS along with an extra OR gate. Because of its *borrow-in* ($B_{in}$) capability, a FS enables cascading, facilitating multi-bit subtraction.

**Figure 5.6:** Diagram and truth table of a FS



| $x$ | $y$ | $B_{in}$ | D | $B_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Adapted from: Bhagat:81.

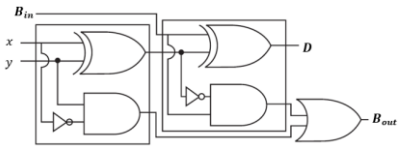
Taking the example of the subtraction $11111_2 - 10010_2$, Figure 5.7 below explains this operation in greater detail.

Each subtractor operates with two bits only. Thus, each stage of subtraction needs a FS to be performed on (except the fist, that can be performed with a HS). The $B_{out}$ of one FS will be the $B_{in}$ of the following FS until the operation comes to an end and the intended result is obtained.

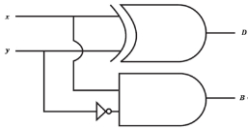
**Figure 5.7:** Binary subtraction using digital circuits.

| | | | | | |
|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 |
| - | 1 | 0 | 0 | 1 | 0 |
| | 0 | 1 | 1 | 0 | 1 |

**A.** We start the operation without a *Borrow* or with *Borrow*$= 0$. For this reason, our circuit can begin with a HS or with a FS with $B_{in} = 0$.




or



| $x$ | $y$ | $B_{in}$ | D | $B_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**B.** The operations will proceed according to the respective truth table.
**HS**: $D = 1 - 0 = 1, B = 0$.
**FS**: $D = 1 - 0, B_{in} = 0 \rightarrow D = 1, B_{out} = 0$

| $x$ | $y$ | D | B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

| $x$ | $y$ | $B_{in}$ | D | $B_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 |
| - | 1 | 0 | 0 | 1 | 0 |
| | 0 | 1 | 1 | 0 | 1 |

**C.** To perform the next operation in our subtraction, we have a FS with a $B_{in} = 0$ (from our previous operation), and we perform $1 - 1, B_{in} = 0$.
According to its truth table, the FS outputs: $D = 0$ and $B_{out} = 0$.

**D.** And the process continues until the operation is complete.

## 5.3 Decoders and Encoders

Decoders and encoders play a vital role in data conversion and signal processing within digital systems. For an examination of these components please refer to Appendix VI. This appendix offers a more complete analysis and examples that will enhance the understanding of how decoders and encoders contribute to efficient data management and system performance.

## 5.4 Conclusion

In conclusion, combinational circuits serve as the cornerstone of digital circuit design, offering powerful capabilities for performing a wide range of logical and arithmetic operations. Through the strategic arrangement of basic logic gates, combinational circuits can be crafted to execute complex functions with precision and efficiency. From basic adders and subtractors to more intricate multiplexers, decoders and encoders, these circuits form the backbone of modern digital systems, facilitating tasks ranging from simple data manipulation to sophisticated signal processing.

# CHAPTER 6 : SEQUENTIAL CIRCUITS

## 6.1 Introduction to Sequential Circuits

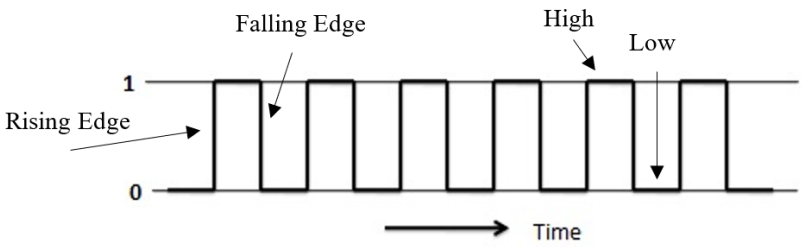
In this chapter, sequential circuits, which, unlike combinational circuits, depend on both current and previous inputs, are explored. This dependency introduces the concept of memory into digital systems, allowing for more dynamic and responsive behaviors. This Chapter also delves into components such as latches, flip-flops, and binary counters, which are essential for storing and processing information in digital systems. These circuits illustrate the traditional applications of Boolean algebra in creating memory elements and controlling system states in real-world devices.

A big challenge for combinational circuits is the lack of a storage mechanism, i.e., they operate without memory. When input values change, the output responds immediately without retaining any information about past inputs and, therefore, alterations in input directly and promptly affect output values. In contrast, sequential circuits depend on both current inputs and previous states, incorporating memory elements like latches and flip-flops to retain past information.

Latches alter their outputs immediately after a change in the inputs. For this reason they are referred to as having transparent outputs. Additionally, they are also level-sensitive, i.e., their outputs can change continuously based on the input if the signal is active.

On the other hand, flip-flops change their outputs only when there is a transition in the clock signal. They are edge-triggered and update their outputs only when there is a specific transition in the clock signal, usually from Low to High or High to Low. Both are essential components in digital electronics for storing data temporarily. While they serve similar purposes, they differ in their behavior regarding when they update their outputs.

**Figure 6.1:** Example of a clock signal and discrete instances of time.



These circuits are categorized into synchronous and asynchronous types: *synchronous* circuits use a common clock signal to update states at regular intervals, ensuring predictable timing, while *asynchronous* circuits transition based on changes in inputs without a global clock (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). A clock in sequential circuits generates pulses at a specific rate, known as the clock cycle time and determines when to update the circuit's state.

## 6.2 Latches

Being level-sensitive devices, i.e., their outputs can change continuously based on the input as long as the enable signal is active, makes them suitable for applications where continuous updates to the stored data are necessary.

There are several types of latches, with the most common ones being the SR latch (*Set-Reset latch*), D latch (*Data latch*), and gated latch (*Enable latch*) (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

The SR latch has two inputs Set (S) and Reset (R). When the S input is asserted (logic High), the latch sets its output to logic High (1). Conversely, when the R input is asserted, the latch resets its output to logic Low (0). However, if both S and R inputs are asserted simultaneously, it can lead to undefined behavior, also known as a *metastable state* (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). This is illustrated in Table 6.1 below, the truth table of a SR Latch with NOR gates.

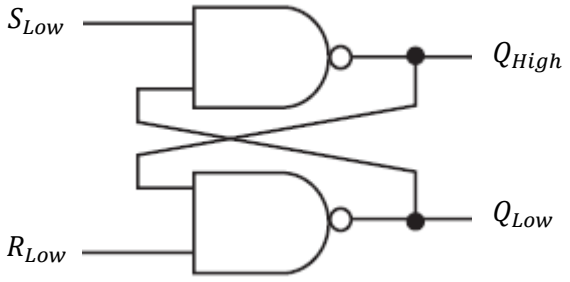
As seen in Table 6.1 below, for a latch with NOR gates, when $S = 0$ and $R = 1$, $Q_{n+1}$ is forced to $0$, representing the state R. On the other hand, when opposite happens, $Q_{n+1}$ is forced to be $1$, i.e., S. If both $S$ and $R = 0$, $Q_{n+1}$ maintains the previous state, $Q_n$, i.e., HOLD.

**Table 6.1:** Truth table of a SR Latch with NOR gates.

| $S$ | $R$ | $Q_{n+1}$ | $\overline{Q_{n+1}}$ | State[13] |
|---|---|---|---|---|
| 0 | 0 | $Q_n$ | $\overline{Q_n}$ | HOLD |
| 0 | 1 | 0 | 1 | RESET |
| 1 | 0 | 1 | 0 | SET |
| 1 | 1 | U | U | *Not used*[14] |

A different case is the SR Latch built with NAND gates, as shown in Figure 6.2 below. In this case, the latch is activated ($Q_{n+1_{High}} = 1$), when the $S = 0$, the input logical state that will impose the NAND result[15] ($Q_{n+1_{High}} = 1$) (Mano & Kime, 2014).

**Figure 6.2:** Example of SR Latch with NAND gates and its truth table.




| $S$ | $R$ | $Q_{n+1_{High}}$ | State |
|---|---|---|---|
| 0 | 0 | U | *Not used* |
| 0 | 1 | 1 | SET |
| 1 | 0 | 0 | RESET |
| 1 | 1 | $Q_n$ | HOLD |

Adapted from: Bhagat:149.

If the activation input of a synchronized latch is connected to the clock signal, its state is continuously updated while the clock is High (1). However, since it is not ensured that the state of the latches remains stable during the High phase of the clock signal, it is also not guaranteed that all latches change synchronously in a complex circuit. Therefore, latches have

[13] The output of a memory element is called ***state***.

[14] The state $R = 1$ *and* $S = 1$ in a SR Latch with NOR gates, as well as the state $R = 0$ and $S = 0$ in a SR Latch with NAND gates, is not normally used and it is considered by some authors as invalid (Mano & Kime, 2014).

[15] The SR latch retains memory due to a *feedback loop* between its outputs $Q_n$ and $\overline{Q_n}$, which connects back into the NAND gates. This feedback keeps $Q_{n+1}$ stable (either at 1 or 0), allowing the latch to hold its last state when both inputs S and R are 1. Essentially, this feedback loop locks $Q_{n+1}$ to its last stable value, enabling the latch to function as a memory cell by preserving its output until it's changed by new input values.

very specific applications, particularly in asynchronous circuits, being less complex and faster memory elements.

### 6.3 Flip-flops

A flip-flop is a bistable multivibrator, meaning it has two stable states, typically represented as Low (0) and High (1); and are used to store binary information and maintain state in sequential systems.

Flip-flops change their outputs only when there is a transition in the clock signal, i.e., are edge-triggered. This mode of operation ensures that their state is altered only once during each clock cycle, allowing for nearly the entire clock cycle to be used for generating new values at the inputs.

There are several types of flip-flops, with the most common ones being the SR flip-flop (*Set-Reset flip-flop*), Master-Slave flip-flop and D flip-flop (*Data flip-flop)* (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

The SR flip-flop has two inputs Set (S) and Reset (R). Initially, the state of the flip-flop depends on its previous state or external conditions and, when both the S and R inputs are low ($S = R = 0$), the flip-flop maintains its current state. If the S input is asserted ($S = 1$) while the R input is low ($R = 0$), the flip-flop enters the S state, i.e., $Q = 1$. Conversely, if the R input is asserted ($R = 1$) while the S input is low ($S = 0$), the flip-flop enters the R state, i.e., $Q = 0$. If both S and R inputs are asserted simultaneously ($S = R = 1$), it can lead to an undefined state, often referred to as the invalid state[16].

[16] In practical implementations, measures are taken to prevent both inputs from being High at the same time to avoid this condition (Mano & Ciletti, 2013; Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019).

The truth table of a SR flip-flop with NOR gates is the same as for the SR latch shown in Table 6.1 above. Additionally, the implementation of a SR flip-flop with NAND gates follows the logic illustrated for the SR latch in Figure 6.2 above.

Once the SR flip-flop is S or R, it remains in that state until it receives another input to change its state, making it suitable for memory storage and sequential logic applications.

The Master-Slave flip-flop comprises two flip-flops connected in series: a master flip-flop and a slave flip-flop; and a control signal, often denoted as $C$, which is used to synchronize the operation of the two flip-flops. During the High phase of the control signal ($C = 1$), the master flip-flop is enabled to accept input commands such as SET or RESET. However, it doesn't immediately pass these commands to the output. Instead, it holds onto the input state until the control signal transitions to its low phase ($C = 0$).

When the control signal transitions from 1 to 0, the master flip-flop stops accepting new input commands and transfers the previously stored state to the slave flip-flop. The slave flip-flop then updates its output based on the transferred state. This mechanism ensures that the state changes in the flip-flop occur only at the falling edge of the clock signal, providing stable and synchronized operation.

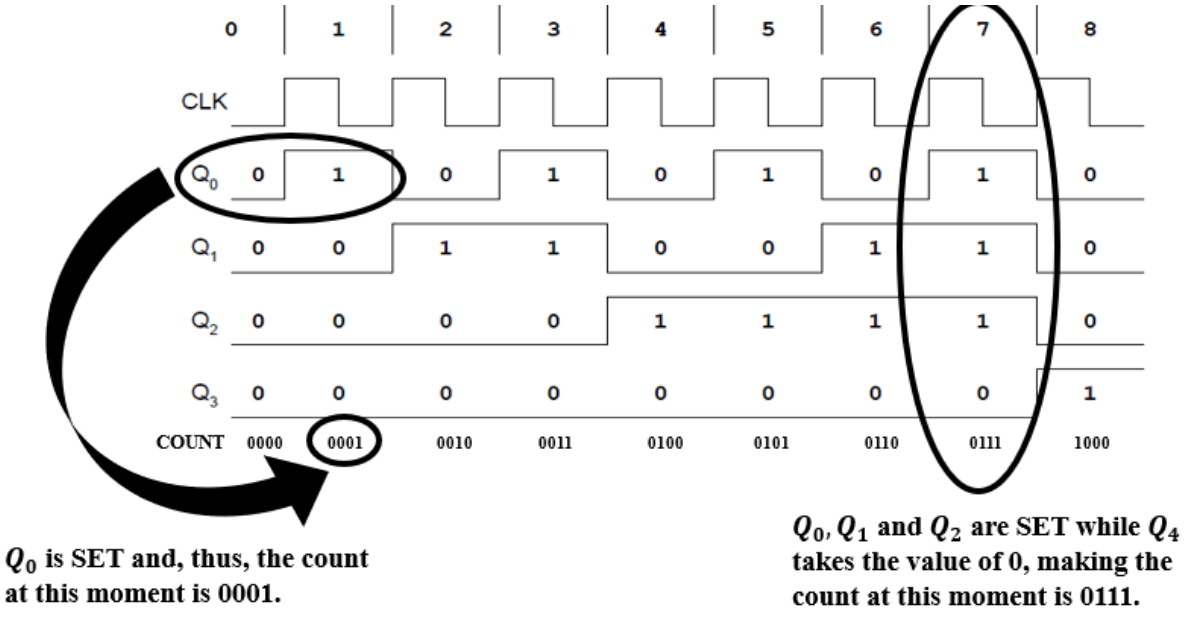
## 6.4 Binary Counter

Registers consist in collections of memory elements, such as latches and flip-flops, organized in a way that allows them to store multiple bits of information simultaneously. They provide a more structured and scalable approach to memory storage compared to individual latches or flip-flops. As such, they play a crucial role in processors, memory units, and other components of digital systems, and contribute to the overall functionality and performance of the system.

A binary counter is a register that counts in binary sequence. It consists of a set of flip-flops connected in a cascaded arrangement, with each flip-flop representing one bit of the

counter. When triggered by clock pulses, the binary counter advances through a sequence of binary states, increasing by one with each clock pulse, as illustrated in Figure 6.3 below.

**Figure 6.3:** Example of a binary counter.




Adapted from: Mano & Kime, 2014.

A good example is a 4-bit binary counter which is a digital circuit used to count the number of clock pulses in binary, typically from 0 to 15 ($2^4 - 1$). This counter uses flip-flops to store and increment the binary count.

A 4-bit binary counter consists of four flip-flops, each representing one bit of the binary count. Each flip-flop can store a binary value of 0 or 1. The flip-flops are connected in such a way that the output of one flip-flop serves as the clock input for the next flip-flop in the series.

All flip-flops are initially set to 0, representing the binary number 0000. The counting mechanism is processed in such a way that the least significant bit (LSB) flip-flop ($Q_0$) toggles[17] its state with each clock pulse and each subsequent flip-flop ($Q_1, Q_2, Q_3$) toggles its

[17] In digital electronics, the term t*oggle* refers to the action of changing the state of a flip-flop or a binary signal from one state to the opposite state. Specifically, if a signal or a flip-flop is in the 0 state, toggling it would

state when the preceding flip-flop transitions from 1 to 0 (a *falling edge*). A detailed explanation of its counting mechanism is given in Figure 6.4 below.

**Figure 6.4:** Binary counter.

serving as the clock input for the next.

| Clock Pulse | State | $Q_3\ Q_2\ Q_1\ Q_0$ |
|---|---|---|
| **0** | ***Initial State*** | **0000** |
| **1** | $Q_0$ toggles | 000**1** |
| **2** | $Q_0$ toggles, causing $Q_1$ to toggle | 00**10** |
| **3** | $Q_0$ toggles | 00**11** |
| **4** | $Q_0$ toggles, causing $Q_1$ and $Q_2$ to toggle | 0**100** |
| **5** | $Q_0$ toggles | 0**101** |
| **6** | $Q_0$ toggles, causing $Q_1$ to toggle | 0**110** |
| **7** | $Q_0$ toggles | 0**111** |
| **8** | $Q_0$ toggles, causing $Q_1, Q_2\ Q_3$ to toggle | **1000** |
| **9** | $Q_0$ toggles | **1001** |
| **10** | $Q_0$ toggles, causing $Q_1$ to toggle | **1010** |
| **11** | $Q_0$ toggles | **1011** |
| **12** | $Q_0$ toggles, causing $Q_1$ and $Q_2$ to toggle | **1100** |
| **13** | $Q_0$ toggles | **1101** |
| **14** | $Q_0$ toggles, causing $Q_1$ to toggle | **1110** |
| **15** | $Q_0$ toggles | **1111** |
| **16** | At the 16th clock pulse, all flip-flops reset to **0000**, and the counting cycle resets. | |

Adapted from: Mano & Kime, 2014.

change its state to 1, and if it is in the 1 state, toggling it would change its state to 0. Toggle flip-flops are used in binary counters, where each flip-flop represents a binary digit. The LSB (bit with the lowest value) toggles at every clock pulse, and each subsequent bit toggles when the previous bit transitions from 1 to 0.

## 6.5 Conclusion

In conclusion, sequential circuits are pivotal in digital systems, as they incorporate memory elements to handle both current inputs and historical data. This chapter has explored the fundamental components of sequential circuits, including latches and flip-flops, which store and manage data, and binary counters, which track numerical values. As such, these circuits find applications in various digital systems, including counters, registers, memory units, finite state machines, and control units of microprocessors.

Understanding these elements and their operation is essential for designing systems that require state retention and timing control. By mastering these concepts, engineers can develop more complex and reliable digital systems that effectively utilize both memory and timing functions.

# CHAPTER 7 : NEURAL NETWORKS AND DEEP LEARNING

## 7.1 Introduction to Deep Learning

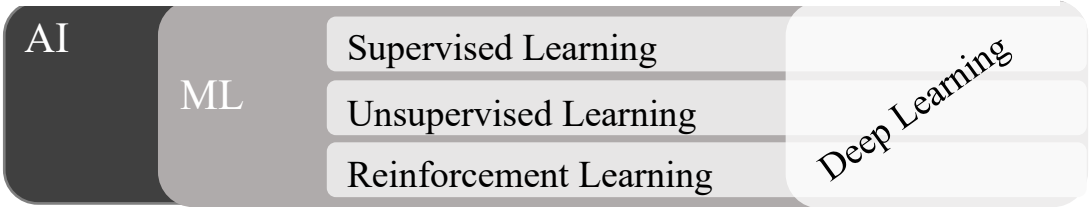
While the first part of this dissertation focuses on classical digital systems and Boolean algebra, the second part transitions into the realm of Machine Learning (ML) and artificial neural networks (NNs). In this chapter, the fundamentals of deep learning, a subfield of ML inspired by the structure and function of the human brain, is introduced. It will begin with the perceptron, a simple NN model, and then it will explore more complex architectures such as Feed Forward Neural Networks (FFNs) and the role of gradient-driven optimization in the training process of these networks.

Artificial intelligence (AI) involves the development of systems capable of performing tasks that typically require human intelligence. This field includes a variety of approaches, such as those using logic, search techniques, and probabilistic methods. A key area within AI is ML, which focuses on making data-driven decisions by constructing mathematical models. This subset has experienced rapid growth and is often, though incorrectly, used interchangeably with the broader term AI (Prince, 2023).

One prominent type of ML model is the deep neural network, and the practice of training these networks on data is known as deep learning. Currently, deep learning models are among the most powerful and widely used in practical applications. For example, translating text between languages often involves natural language processing algorithms (NLP), identifying objects in images typically uses computer vision systems, and interacting with digital assistants frequently relies on speech recognition technology—all powered by deep learning (Prince, 2023).

ML techniques can generally be classified into three categories: supervised learning, unsupervised learning, and reinforcement learning. Today, cutting-edge developments in each of these areas heavily depend on deep learning.

**Figure 7.1:** Introduction to deep learning.




Adapted from: Prince, 2023.

A model in ML can be thought of as a collection of equations that map inputs to outputs. The specific equation, or curve, that best fits the data is determined using *training data* — examples of input and corresponding output. When it is said that one is *training or fitting a model*, it means that one is searching through the possible equations to find the one that best matches the training data. For example, a music classification model would require many audio clips, each labeled with its genre by a human expert. These labeled pairs serve as a guide during the training process, hence the term *supervised learning* (Prince, 2023).

On the other hand, building a model using input data without corresponding output labels is known as *unsupervised learning*. Without output labels, there is no "supervision" to guide the learning process. Rather than focusing on mapping inputs to outputs, the goal is to investigate and comprehend the inherent structure of the data. Just like with supervised learning, the nature of the data can differ significantly; it might be discrete or continuous, range from low to high dimensionality, and have consistent or varying lengths.

In contrast, *reinforcement learning* involves an agent learning to make decisions by interacting with an environment and receiving feedback in the form of *rewards*. The goal is to learn which next action leads to maximizing the sum of the rewards resulting in the computation of an *optimal policy*. For instance, a robot learning to walk might try different movements, receiving positive feedback when it stays upright and negative feedback when it falls. Over time, the robot learns which actions lead to better outcomes, even without explicit labeled examples.

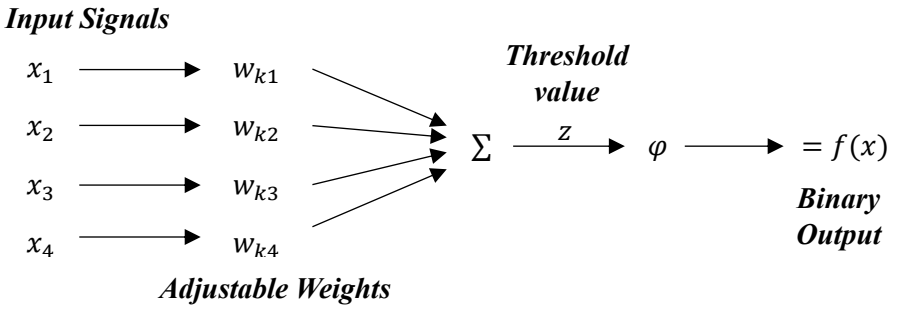
This dissertation focuses on NNs, a type of deep learning architecture which composes highly flexible supervised ML models designed to handle a broad spectrum of input-output relationships. NNs simplify the process of identifying the best-fit relationship for training data and excel at managing inputs that can be large, of varying lengths, and complex in structure. They can generate different types of outputs, including single real numbers (for regression tasks), multiple numbers (for multivariate regression), or probabilities for multiple categories (for classification tasks).

This chapter starts with an introduction of the fundamental building block of a NN, the *perceptron*, it will then cover a more complex and recent architecture, the FFN, and the algorithms used for the training process. Here the basis and the fundamental concepts for the next chapter, that consists on the implementation of a specific type of NN, the Logic Gate Neural Networks (LGNs), are laid.

## 7.2 The Perceptron

In 1943, McCulloch and Pitts introduced the concept of an *artificial neuron* that could combine inputs to produce an output, though their model lacked a practical learning algorithm. Later, in 1958, Rosenblatt advanced this idea with the perceptron, a model that linearly combined inputs and used a threshold to make binary decisions, illustrated in Figure 7.2 below, and an algorithm to adjust the weights based on data, shown in (7.1).

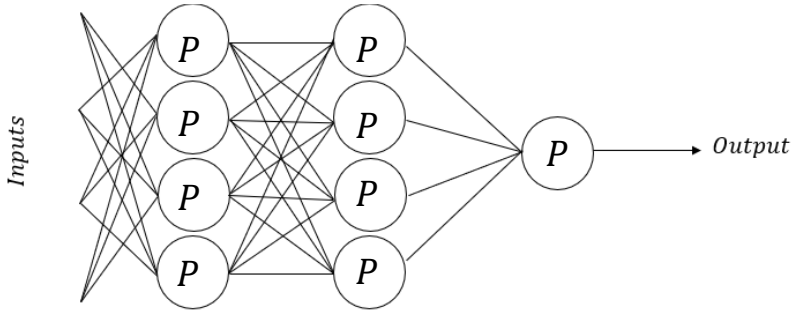
**Figure 7.2:** Example of a representation of the perceptron with four inputs.




$$\boldsymbol{\varphi} = \begin{cases} 0, & if \sum_j w_j x_j \leq threshold\ value \\ 1, & if \sum_j w_j x_j > threshold\ value \end{cases} \tag{7.1}$$

The perceptron processes multiple binary inputs $x1, x2, \ldots$ and produces a single binary output, $f(x)$, based on weighted inputs and a threshold. The output is determined by whether the weighted sum exceeds the threshold, as shown in (7.1). The threshold can be adjusted to create different models, and the weights indicate the importance of each input.

**Figure 7.3:** Example of a NN with perceptrons (P).




Adapted from: Nielsen, 2015.

Various perceptrons can be connected in series and parallel to form a NN, as shown in Figure 7.3 above. In a NN, perceptrons in the first layer make decisions by applying the model from (7.1) above, while perceptrons in subsequent layers make more complex decisions based on outputs from previous layers. Each perceptron has a single output, which serves as input for multiple perceptrons in the next layer.

This model can be simplified: the weighted sum can be expressed as a vectorial product (7.2) below, in which $\boldsymbol{w}$ and $\boldsymbol{x}$ are the weights and the input vectors respectively. The threshold can also be replaced by a bias term, $\boldsymbol{b}$, (7.3) and (7.4) below. This bias term "adjusts" how easily a perceptron produces an output of 1.

$$\sum_j w_j x_j = \boldsymbol{w} \cdot \boldsymbol{x} \tag{7.2}$$

$$\boldsymbol{w} \cdot \boldsymbol{x} \leq threshold\ value \leftrightarrow \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b} \leq 0 \tag{7.3}$$

$$\boldsymbol{P_{output}} = \begin{cases} \boldsymbol{0}, & \boldsymbol{if\ w \cdot x + b \leq 0} \\ \boldsymbol{1}, & \boldsymbol{if\ w \cdot x + b > 0} \end{cases} \tag{7.4}$$

The perceptron, despite its initial promise for linearly separable problems, faced significant limitations in handling more complex classification tasks. It could only effectively classify linearly separable data and struggled with intricate patterns and multiple classes.

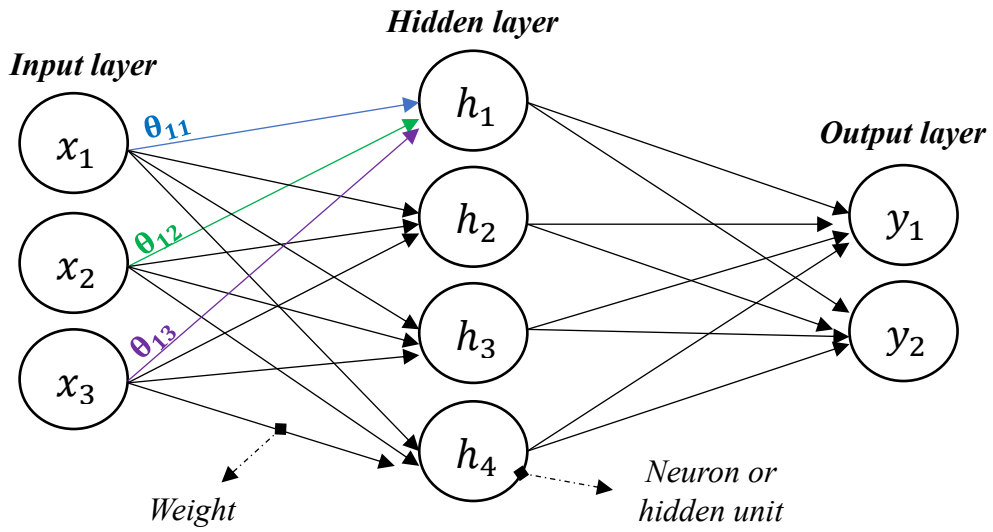
These issues, along with ineffective training methods for multi-layer networks and unrealistic expectations, led to reduced interest and funding in NN research.

In 1969, Minsky and Papert highlighted these limitations and proposed that incorporating *hidden layers* with nonlinear *activation functions*, forming what is now known as a *multilayer perceptron* (MLP), could address more complex relationships between inputs and outputs, as it will be explained next. However, they found Rosenblatt's original algorithm inadequate for training these advanced models and it wasn't until the 1980s that practical algorithms, such as *backpropagation*, were developed, which revitalized interest and progress in NNs.

### 7.3 General Concepts

NNs come with a lot of specialized terminology which is important to understand in order to capture the general concepts. In NNs architecture, different sections are often referred to as *layers*. Figure 7.4 below shows a representation of an FFN with three *inputs*, one *hidden layer* with four *hidden units* and two *outputs*.

**Figure 7.4:** FFNs.




Adapted from: Prince, 2023:35.

Networks with only one hidden layer are known as *shallow neural networks* (SNNs), whereas those with multiple hidden layers are called *deep neural networks*.

In SNNs, connections between layers are directed from the input layer to the hidden layer and then to the output layer, i.e., they form an acyclic graph, which is why they are called FFNs. When every neuron in one layer is connected to every neuron in the next layer, the network is *fully connected*.

Each hidden unit receives the values from all other hidden units in the layer before. It multiplies them with their weights (represented by the connections in Figure 7.3 above), adds them together and applies a "so-called" activation function to the result. The procedure can be represented mathematically in (7.5) as an example for the output of the hidden unit $h_1$.

$$h_1 = a[\theta_{11}x_1 + \theta_{12}x_2 + \theta_{13}x_3 + \theta_{10}] \tag{7.5}$$

Where a[•] is the activation function and $\theta_{ji}$ is the weight applied to the input $x_i$ for the hidden unit $h_j$, except for $\theta_{10}$, which is the bias of $h_1$. Each hidden unit $h_j$ has its own bias term, $\theta_{j0}$ (not represented in the figure). The weights $\theta_{ij}$ differ from connection to connection between layers and not all of them are represented in Figure 7.4. As it will be seen ahead, these values, are better represented by a matrix $\boldsymbol{\Omega}_{\mathrm{k}}$ with dimensions $D_k \times D_{k-1}$ where $D_{k-1}$ is the number of hidden units (or inputs if it is a first layer) in the layer before and $D_k$ is the number of hidden units (or output if it is the last layer) in the layer after. We'll later explore the mathematical equations for this architecture as well as the role of the activation function in greater detail.

After understanding the general concept and terminology of a NN, it can now be explored how to train these models. *Learning or training a model*, means finding a set of parameters, $\boldsymbol{\phi}$, that allow the model to make accurate predictions from given inputs. In FFNs the set $\boldsymbol{\phi}$ is composed of all the weights and biases of the network. To train an FFN, a *training dataset,* consisting of pairs of inputs and outputs $\{\boldsymbol{x_i}, \boldsymbol{y_i}\}$ to learn these parameters, is used.

The goal is to choose parameters $\boldsymbol{\phi}$ that map a set of inputs to its corresponding output as *accurately* as possible. For a given input, $\boldsymbol{x_i}$, the discrepancy between what the network

outputs, $\hat{\boldsymbol{y}}_i$, and the true value (also called *ground truth*), $\boldsymbol{y}_i$, is measured by what it's called a *loss function*[18], $L[]$, which provides a single value summarizing how well the model predicts the outputs based on the inputs for a given set of parameters, $\boldsymbol{\phi}$.

The loss function can be seen as a function of these parameters, $L[\boldsymbol{\phi}]$. Training the model involves finding the parameters $\widehat{\boldsymbol{\phi}}$ that minimize this loss function as in (7.6).

$$\widehat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{argmin}\, L[\boldsymbol{\phi}] \tag{7.6}$$

After training, that is, after finding the minima of $L$, the model's performance needs to be evaluated by testing it on a separate dataset to ensure it can *generalize* well to new, unseen examples. Adequate performance[19] on this test data indicates readiness for deployment.

As complicated as just minimizing the loss function is, some other typical problems may arise thereafter which make training a model an arduous task. For instance, if a too shallow of a network is taken or global minima of the loss function is not reached, the model may not capture the true relationship between input and output, resulting in *underfitting*. On the other hand, an overtrained and/or overly complex model may learn from the test data *too well*, such that it begins to fit the noise in the data leading to a poor generalization, a problem known as *overfitting*.

To know when to stop training to avoid overfitting, a *validation dataset* is typically used. This dataset is independent from the training and test sets and contains separate data. A continuous performance evaluation is computed on the validation set during training. A basic strategy starting point is to stop training when the loss on the validation set reaches a minimum.

[18] In ML the terms "*loss function*" and "*cost function*" are often used interchangeably. However, a loss function refers to the individual term associated with a single data point, whereas the cost function refers to the overall quantity that is to be minimized, encompassing all the individual loss terms. A cost function may also include additional terms not directly linked to individual data points (Prince, 2023).

[19] The performance of a model may be defined in various ways depending on the implementation goal. Various functions that measure the performance include for instance MSE (mean squared error), MAPE (mean average percentage error) for regression problems or Precision and Recall for classification problems.

Fitting problems can be overcome by correctly dimensioning the network to be trained. However, the number of layers and the number of hidden units in each layer are not parameters of the network. These are called *hyperparameters* instead. Hyperparameters are not known *a priori* and heavily depend on the complexity of the relationship between input and output. It is typical to train various networks of different sizes with the same data and evaluate them separately in what is called the *hyperparameter search*.

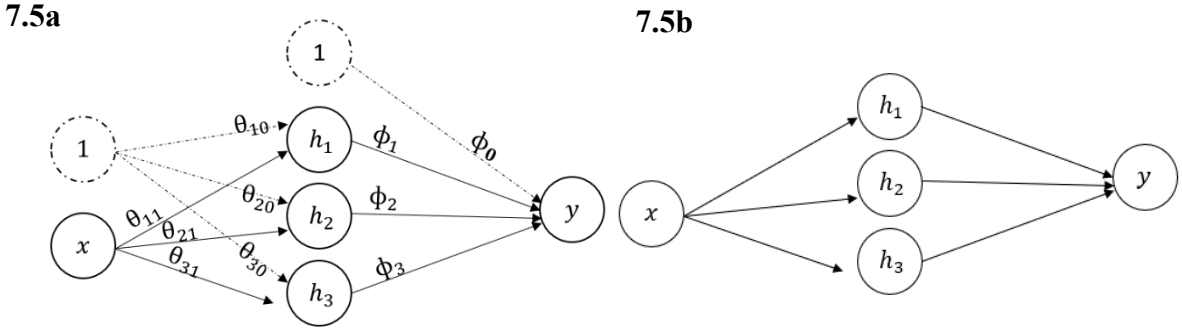
Building on these concepts, it is important to capture the mathematical representation of a FFN.

## 7.4 Feed Forward Networks (FFNs)

NNs are in their essence mathematical functions $\boldsymbol{y} = f[\boldsymbol{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that map multivariate inputs, $\boldsymbol{x}$, to multivariate outputs, $\boldsymbol{y}$,. Taking the example of an FFN that maps a scalar input $x$ to a scalar output $y$ and includes ten parameters $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$ (7.7). This network is represented in Figure 7.5 below.

$$y = f[x, \phi] = \phi_0 + \phi_1 a[\theta_{10} + \theta_{11}x] + \phi_2 a[\theta_{20} + \theta_{21}x] + \phi_3 a[\theta_{30} + \theta_{31}x] \qquad (7.7)$$

**Figure 7.5:** Example of an FFN.




Adapted from: Prince, 2023: 29.

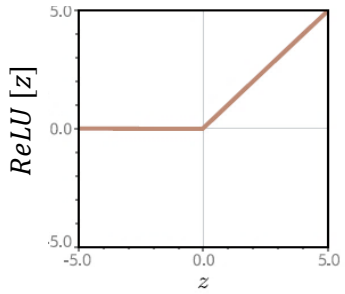
This calculation can be explained in three steps. First, the three linear functions of the input $x$ are computed: $\theta_{10} + \theta_{11}x$, $\theta_{20} + \theta_{21}x$ and $\theta_{30} + \theta_{31}x$. Then, an *activation function*

a[•] is applied to each of these results. Finally, these three activations are weighted with $\phi_1, \phi_2$ and $\phi_3$, are summed and an offset $\phi_0$ is added (Prince, 2023). This is called shallow network (SNN) as it possesses only one hidden layer.

There are many activation functions a[•] that can be used, but one of the most common choices is the rectified linear unit or ReLU (7.8). This function returns the input when it is positive and zero otherwise, as shown in Figure 7.6 below.

$$a[z] = ReLU[z] = \begin{cases} 0, & z \leq 0 \\ z, & z > 0 \end{cases} \tag{7.8}$$

**Figure 7.6:** Graphic representation of the ReLU activation function.




Source: Prince, 2023:26.

In (7.7) above, a family of functions is represented, with the specific function determined by the ten parameters in $\phi$. If these parameters are known, $y$ can be predicted for a given input $x$ in the so-called *inference* step by evaluating the equation. Given a training dataset $\{x_i, y_i\}$ with $i = 1 \ldots N$, where $N$ is the number of observations, a loss function $L[\boldsymbol{\phi}]$ can be defined to measure how well the model fits the data for any set of parameters $\boldsymbol{\phi}$. As mentioned before, training the model involves finding the parameter values $\hat{\boldsymbol{\phi}}$ that minimize this loss. For regression problems like this example the loss function is typically a variation of the least squares[20] given in (7.9), where $\hat{y}_i$ is the output of the network for a given set of $\boldsymbol{\phi}$ and $x$ and $y_i$ is the actual observed value from the training dataset (*ground truth*). The index $i$ denotes the $i^{th}$ sample from the training data. So, the loss function is merely an equation that

[20] The least squares equation is the natural outcome when applying the Method of Maximum Likelihood assuming errors of constant variance and zero expected value.

sums the squares of the difference, i.e., the errors, between what the network predicts and the true observed value.

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \tag{7.9}$$

The concept of this example can be generalized to a network with $D$ hidden units, where the $d^{th}$ hidden unit is defined as (7.10). These hidden units are then combined linearly to produce the output (7.11) below.

$$h_d = a[\theta_{d0} + \theta_{d1}x] \tag{7.10}$$

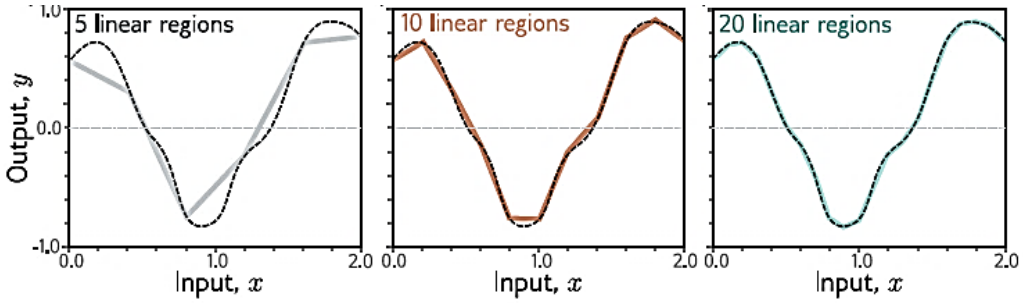
$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d \tag{7.11}$$

The number of hidden units in a NN determines its *capacity*[21] (Prince, 2023). With ReLU activation functions, for a SNN, the output of a network with $D$ hidden units can have at most $D$ joints, making it a piecewise linear function with up to $D + 1$ linear regions. As more hidden units are added, the model can approximate increasingly complex functions.

In fact, with sufficient capacity, i.e., enough hidden units, a NN with just one layer can approximate any continuous 1-dimensional function defined on a compact subset of the real line to an arbitrary degree of precision (Prince, 2023). Each additional hidden unit introduces another linear region to the function. As the number of these linear regions increases, they represent smaller and smaller segments of the function, which can be closely approximated by a line, as seen in Figure 7.7. According to the *universal approximation theorem*, a SNN with sufficient hidden units can approximate any continuous function defined on a compact subset of $\mathbb{R}^D$ to arbitrary precision.

[21] While the "capacity" of NN still lacks a universal formal definition, it broadly refers to a network's ability to fit a wide variety of functions or data patterns. In other words, it indicates the model's flexibility or expressiveness: a high-capacity model can learn more complex patterns, while a low-capacity model is more limited in this regard. It depends on factors like the number of layers, neurons, and connections within the network.

**Figure 7.7:** Universal approximation theorem.

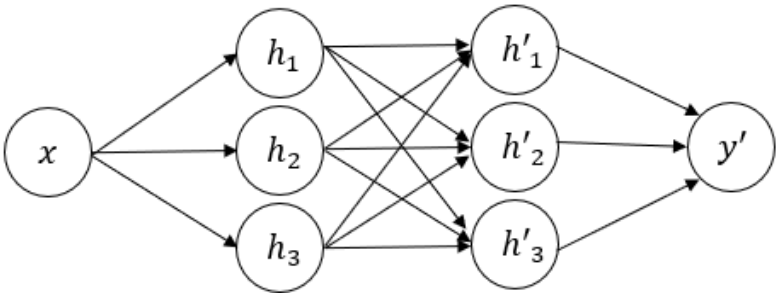



Source: Prince, 2023:30.

This theorem also holds to networks that map multivariate inputs $\boldsymbol{x} = [x_1, x_2, \ldots, x_{D_i}]^T$ to multivariate outputs $\boldsymbol{y} = [y_1, y_2, \ldots, y_{D_o}]^T$.

As shown, with enough hidden units, SNNs can approximate any continuous function in high-dimensional space. However, some functions require an impractically large number of hidden units for a SNN to approximate accurately. Deep networks, with more than one hidden layer, on the other hand, can generate many more linear regions with the same number of parameters, making them more efficient for describing a wider range of functions. In Figure 7.8 a network with two hidden layers is represented.

**Figure 7.8:** Deep network with two hidden layers, each containing three hidden units.




Adapted from: Prince, 2023:45.

Modern networks might have over a hundred layers with thousands of hidden units per layer. The number of hidden units in each layer is known as the network's *width*, while the number of hidden layers is its *depth*.

The number of layers is denoted as $K$ and the number of hidden units in each layer as $D_1, D_2, \ldots, D_K$. These are hyperparameters, which are set before training the model parameters (the *weights* and *biases*) as mentioned before.

It can be easily understood that for NN with many layers, the notation can become quite complex. Therefore, we'll simplify things by denoting the vector of hidden units at layer $k$ as $\boldsymbol{h}_k$. $\boldsymbol{\beta}_k$ will be used for the biases that affect the hidden layer $k+1$ and $\boldsymbol{\Omega}_k$ for the weights applied to the $k^{th}$ layer that influence the $(k+1)^{th}$ layer. This allows us to express a general deep network $\boldsymbol{y} = f[\boldsymbol{x}, \boldsymbol{\phi}]$ with $K$ layers more succinctly as (7.12).

$$\begin{aligned}
\boldsymbol{h}_1 &= \mathrm{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \boldsymbol{x}] \\
\boldsymbol{h}_2 &= \mathrm{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \boldsymbol{h}_1] \\
\boldsymbol{h}_3 &= \mathrm{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \boldsymbol{h}_2] \\
&\dots \\
\boldsymbol{h}_K &= \mathrm{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \boldsymbol{h}_{K-1}] \\
\boldsymbol{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \boldsymbol{h}_K
\end{aligned} \tag{7.12}$$

The model's parameters $\boldsymbol{\phi}$ include all the weight matrices and bias vectors, expressed as $\boldsymbol{\phi} = \{\boldsymbol{\beta}_K, \boldsymbol{\Omega}_K\}_{k=0}^{K}$. For each layer $k$ with $D_k$ hidden units, the bias vector $\boldsymbol{\beta}_{K-1}$ will be of size $D_k$, while the final bias vector $\boldsymbol{\beta}_K$ will have the size $D_o$, corresponding to the output dimensions. The weight matrix $\boldsymbol{\Omega}_0$ connects the input size, $D_i$, to the first hidden layer size $D_1$, making its dimensions $D_1 \times D_i$. The last weight matrix $\boldsymbol{\Omega}_K$ connects the last hidden layer size $D_K$, to the output size $D_o$, making its dimensions $D_o \times D_K$. All other weight matrices $\boldsymbol{\Omega}_k$ have dimensions $D_k \times D_{k-1}$. The network can be concisely represented as a single function (7.13).

$$\boldsymbol{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K a\left[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} a\big[\dots \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 a\big[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 a[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \boldsymbol{x}]\big]\dots\big]\right] \tag{7.13}$$

Deep networks offer the benefit of easier training compared to SNNs. This is because moderately deep networks often have a wide range of similar solutions that are easier to find. However, increasing the depth of the network can make training more challenging as it increases the dimensions of the hyperspace and requires computationally more power and memory. Additionally, deep networks generally perform better when applied to new, unseen data due to being able to generalize better with fewer parameters (Prince, 2023).

## 7.5 Gradient-Driven Optimization

As already seen, the loss in a NN is influenced by its parameters, and the goal is to find the parameter values that reduce this loss as much as possible. This process is called *learning* the network's parameters, *training the model* or *fitting the model*. The objective is to adjust the model's parameters to minimize a loss, $L[]$ (7.14) below.

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{argmin}\, L[\boldsymbol{\phi}] \tag{7.14}$$

One common method for this is the *gradient descent*. It starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and follows two steps repeatedly. In the first step, it is calculated how the loss changes with respect to the parameters (known as *computing the gradients*). This gradient shows the direction in which the loss is increasing (*uphill*) (7.15) below. In the second step, the parameters are adjusted using these gradients to minimize the loss (7.16) below i.e., update the parameters are slightly updated in the opposite direction (*downhill*) by a small amount, denoted by $\alpha$, also called the *learning rate*, which is why there's a negative sign in (7.16).

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix} \tag{7.15}$$

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha \frac{\partial L}{\partial \boldsymbol{\phi}} \tag{7.16}$$

The size of each update is controlled by, $\alpha$[22], and can either be fixed or determined by testing different values to see which one reduces the loss the most. When the loss function reaches its minimum, the surface becomes flat, meaning there's no further downhill direction

[22] The learning rate, $\alpha$, is a positive scalar that determines the extent of the adjustment, i.e., determines how much the model's parameters are adjusted in each step during the training process. Advanced training methods and variations to the Gradient Descent (like the Adam) update these parameters dynamically during training to help optimize the minima search.

to move into, and the gradient becomes zero, so the parameters stop changing. In practice, it is recommended to keep an eye on the gradient size and stop the process when it becomes very small.

Gradient descent is not exclusive to NNs. Without loss of generality, it will now be considered the example of using gradient descent for a simple linear regression model $f[\boldsymbol{x}, \boldsymbol{\phi}]$, where a scalar output $y$ is predicted from a scalar input $x$, with parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, which represent the $y$-intercept and the slope of the line (7.17) below.

$$y = f[\boldsymbol{x}, \boldsymbol{\phi}] = \phi_0 + \phi_1 x \tag{7.17}$$

Given a dataset with $I$ pairs of input $x_i$, and output $y_i$, the least squares will be used again as loss function (7.18), where the term $\ell i = (\phi_0 + \phi_1 x_i - y_i)^2$ represents how much the $i^{th}$ training example contributes to the overall loss.

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} (f[x, \boldsymbol{\phi}] - y_i)^2 = \sum_{i=1}^{I} (\phi_0 + \phi_1 x_i - y_i)^2 \tag{7.18}$$

The derivative of the loss function with respect to the parameters can be found by adding up the derivatives from each individual data point (7.19), where the individual derivatives are given in (7.20).

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \frac{\partial}{\partial \boldsymbol{\phi}} \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} \frac{\partial \ell_i}{\partial \boldsymbol{\phi}} \tag{7.19}$$

$$\frac{\partial \ell_i}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial \ell_i}{\partial \phi_0} \\ \frac{\partial \ell_i}{\partial \phi_1} \end{bmatrix} = \begin{bmatrix} 2(\phi_0 + \phi_1 x_i - y_i) \\ 2x_i(\phi_0 + \phi_1 x_i - y_i) \end{bmatrix} \tag{7.20}$$

This means that the overall gradient is a sum of gradients from each individual data point in the training set. Each of these individual gradients is calculated based on how much the current parameters differ from the target value for that specific example. A simpler example of the application of gradient descent is given in Appendix VII.

Gradient descent is a common method for optimizing loss functions, especially in straightforward scenarios. However, it struggles with complex, high-dimensional data because it tends to get stuck in local *minima*, always following the steepest descent path, which might not lead to the best solution. This issue is even more pronounced in NN models with millions of parameters, making it difficult to find the global minimum or even a satisfactory one.

To overcome these challenges, *stochastic gradient descent*, as the name indicates, introduces randomness into the process, allowing for a broader exploration of the loss function and helping to avoid the limitations of standard gradient descent. Thus, the update rule for the model parameters $\boldsymbol{\phi}_t$ at iteration $t$ as described in (7.16) above, is now given by (7.21) below, where $\boldsymbol{\mathcal{B}}_t$ is a set containing the indices of the input/output pairs in the current *batch* and, as previously explained for gradient descent, $\ell_i$ is the loss due to the $i^{th}$ pair.

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \tag{7.21}$$

In standard gradient descent, the gradient is calculated using the entire training dataset at each step, which can be computationally heavy, especially for large datasets. In contrast, stochastic gradient decent randomly selects a subset of the training data at each step, known as a *minibatch* or *batch*, to compute the gradient. The update rule adjusts the parameters by moving them in the direction that reduces the loss, based on the data in the current batch. The amount of this adjustment is controlled by the learning rate. This significantly reduces the computational load, making the process faster.

Typically, these batches are selected without replacement, meaning once a data point is used, it's not reused until all data points have been utilized. When the algorithm has gone through all the data once, this is called an *epoch*, and it starts over with the full dataset. The size of a batch can vary, from a single data point (the smallest possible batch) to the entire dataset (which is known as *full-batch gradient descent* and is the same as standard gradient descent).

Stochastic gradient descent offers several advantages: it updates the model efficiently by using random subsets of data, making it less computationally expensive and potentially avoiding local *minima* and saddle points[23]. Although it introduces some randomness, this method helps the model fit the data well and can improve generalization to new data. Typically, stochastic gradient descent starts with a higher learning rate to explore the parameter space and then gradually reduces the rate to fine-tune the model as training progresses.

Most common libraries for training NN (*TensorFlow, PyTorch* etc.) already implement the stochastic gradient descent out of the box with little to no prior configuration. However, most of them already apply significant improvement techniques such as Momentum, Adam etc.

For each gradient descent iteration, the derivatives of the loss function with respect to the thousands or even millions of possible parameters for a NN must be computed. This imposes a tremendous computational problem. To efficiently compute the derivatives of the network, the *backpropagation* algorithm is usually employed. The strategy involves taking advantage of the architecture of the network and the mathematical patterns in the function describing the network.

## 7.6 Backpropagation Algorithm

Building on the concepts of gradient-driven optimization discussed in the previous section, the backpropagation algorithm plays a crucial role in training NNs by systematically updating weights to minimize errors. This algorithm relies on gradient descent to propagate errors backward through the network, adjusting parameters to improve model accuracy. For a

[23] A saddle point on the loss function is a point where the gradient is zero, but it's not a true minimum or maximum. In NNs with many parameters, saddle points are common and can slow down training since the algorithm may struggle to move past these areas with near-zero gradients. Modern optimizers like Adam help avoid getting stuck at saddle points by using momentum from past gradients, enabling faster and more reliable convergence.

more complete exploration of the backpropagation algorithm, including its detailed workings and implementation, please refer to Appendix VIII.

## 7.7 Conclusion

NNs represent a significant advancement in AI, offering powerful tools for solving complex problems. This chapter has covered essential concepts, from the basics of deep learning and perceptrons, to more advanced topics like FFNs, gradient-driven optimization. Understanding these principles is crucial for leveraging the full potential of deep learning in developing intelligent and adaptive systems.

The next chapter will focus on implementing LGNs, applying the theoretical concepts discussed earlier to practical scenarios.

# CHAPTER 8 : LOGIC GATE NEURAL NETWORKS

## 8.1 Introduction

This final chapter presents the practical experiment of this dissertation, the implementation of an LGN. Building upon the principles of digital logic and NNs, LGNs use differentiable logic gates to create networks that mimic the behavior of traditional circuits while benefiting from the adaptability of NNs. Based on the framework introduced in the article *Deep Differentiable Logic Gate Networks*, by Petersen *et al.* (2023), this chapter demonstrates how NNs can be constructed using logical operations as their foundation. This hybrid approach allows for the exploration of new possibilities in circuit design, where ML models are seamlessly integrated with traditional logic-based systems.

Unlike conventional NNs, LGNs use logic functions, making them non-differentiable and traditionally incompatible with gradient-based optimization. To address this, Petersen *et al.* (2023) propose a differentiable relaxation of logic gates, enabling the application of gradient-based algorithms during training. Through this approach, each neuron represents a probability distribution over a list of possible logic functions, allowing the network to learn which logic function is optimal at each point. Following training, the network is converted to a discrete form by selecting the logic function with the highest probability for each neuron. This process yields a sparse and efficient network, where each neuron has only two inputs with two possible states (0 or 1), resulting in rapid inference.

It is important to note that this approach differs from binary neural networks, which retain weighted connections and reduce weights or activations to binary precision. Instead, LGNs lack weights altogether, relying solely on specific logic functions at each neuron. Unlike other sparse networks, LGNs aim to learn the optimal logic function for each neuron, with fixed, randomly initialized connections.

There are sixteen possible logic functions (see Table 8.1 on page 112), as each neuron corresponds to a Boolean function with two inputs and one output. The training process

leverages softmax[24] in each neuron to select the most suitable function. This process will be further explored in the next sections.

## 8.2 Logic Gate Networks (LGNs)

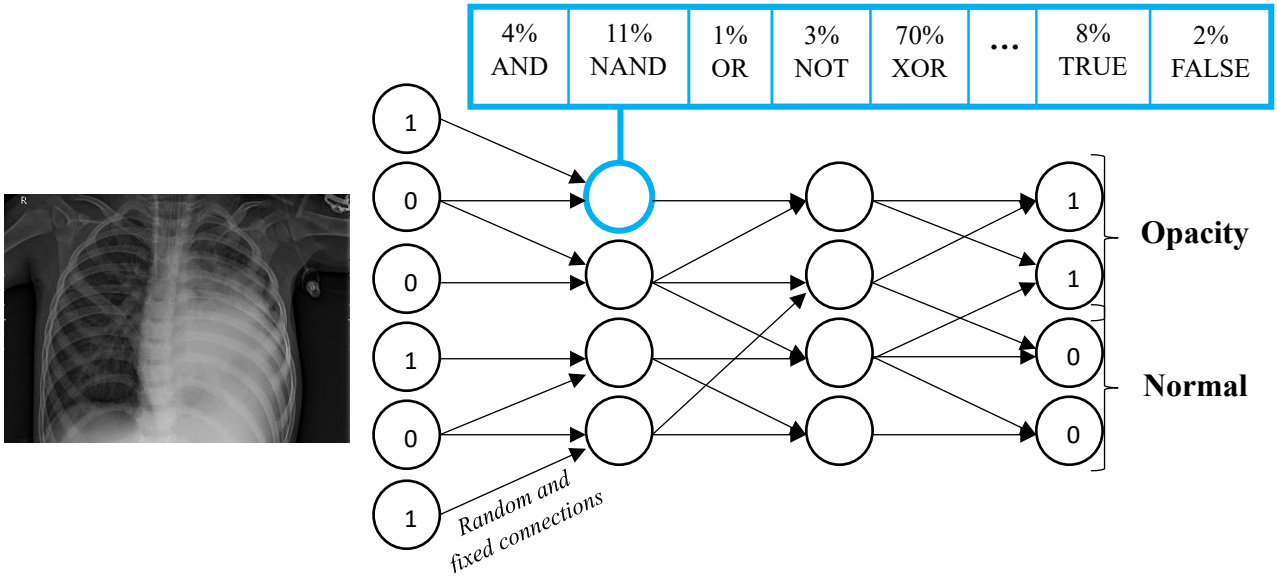
Based on the presented concepts, LGNs resemble traditional NNs but differ fundamentally by representing each neuron as a binary logic function (e.g., AND, NAND, or NOR) rather than through the sum of its weighted connections. Given a *binary input vector*, LGNs process pairs of Boolean values through these logic gates, passing the outputs to subsequent layers. This structure eliminates the need for activation functions, as the logic functions already provide inherent non-linearity.

Instead of weights, LGNs are parameterized by the *selection of logic functions* at each neuron. While a single binary output could theoretically serve for prediction, this would produce ungraded outputs (0 or 1) that would lack nuance. To achieve finer distinctions, LGNs assign multiple neurons *per class* and aggregate their outputs by *summation*, enabling graded predictions. Each neuron thus adds evidence for a particular class, contributing to enhanced classification accuracy.

An example LGN architecture is illustrated in Figure 8.1 below, where each node represents a logic function, and the distribution of logic functions (in blue) represents the differentiable relaxation discussed in the next section.

[24] The softmax function is a mathematical function often used in NNs to transform a set of real values into probabilities. It takes an input vector and converts each element in that vector into a probability score, with all scores adding up to 1. This makes it particularly useful in classification tasks where is important to interpret outputs as probabilities for each class. It takes the form: $softmax(w_i) = \frac{e^{w_i}}{\sum_j e^{w_j}}$, where $w_i$ represents the input score or logit for the $i^{th}$class with each $w_i$ corresponding to the model's raw prediction; and where $w_j$ represents the input scores or logits for all possible $j$ classes. The denominator $\left(\sum_j e^{w_j}\right)$ sums over the exponentials of all logits, ensuring that the softmax output probabilities sum to 1.

**Figure 8.1:** Summary of the differentiable LGN introduced by Petersen *et al.* (2023).




Adapted from: Petersen *et al.*, 2023.

As shown in Figure 8.1 above, the image pixels are transformed into *Boolean inputs*, which are then processed by a layer of neurons, each receiving two inputs. After an initial random initialization of the connections between neurons, these connections remain fixed. Each neuron is represented by a *probability distribution* over possible logic functions, which is learned during training. After training, the most likely *function* for each neuron is selected and used at inference for the classification into either “Opacity” or “Normal”. Multiple neurons contribute to the output for each class, and their results are combined using *bit-counting* to generate class scores. This process will be explained in more depth for our experience in further sections. For clarity, the number of neurons shown in Figure 8.1 is significantly reduced.

LGNs offer significant computational and memory efficiency. Since they rely solely on bitwise operations rather than floating-point arithmetic, LGNs can execute faster and with reduced computational demand. This makes them particularly suitable for real-time applications and low-power environments, such as embedded systems. Furthermore, because LGNs do not require weights, they avoid the memory overhead associated with storing and

updating weight matrices. This sparsity, combined with binary inputs, leads to a much smaller memory footprint compared to dense NNs, making LGNs ideal for applications where memory and storage are constrained.

### 8.3 Differentiable Logic Gate Networks

As mentioned, training LGN poses a great challenge, due to their non-differentiable nature, which prevents the use of traditional gradient-based training algorithms. To overcome this, Petersen *et al.* (2023) propose relaxing LGNs into differentiable versions, enabling gradient-based optimization. This relaxation happens only during the training of the model.

In their approach, binary activations (0 or 1) are replaced with *probabilistic activations* in the range [0, 1]. The standard logic gates are then substituted with operators that compute the expected value based on the input probabilities. For example, the probability of two independent events, $q_1$ and $q_2$, both occurring is the product of their probabilities $(q_1.q_2)$. This approach leverages probabilistic T-norm and T-conorm operations, which are introduced in Appendix VIX.

Following the *probabilistic relaxation* of the logic gates, the activation of a neuron is defined by applying the $i^{th}$ real-valued operator to the inputs $q_1$ and $q_2$, as expressed in (8.1), where $f_i$ is the $i^{th}$ real-valued operator (as shown in Table 1 below) and $q_1$ and $q_2$ represent the inputs to the neuron.

$$a' = f_i(q_1, q_2) \tag{8.1}$$

During training, a *probability distribution* over the 16 possible logic functions is learned for each neuron. This distribution is parameterized by 16 real-valued parameters, according to Table 8.1 below, which are transformed via the *softmax function* to ensure a valid probability distribution (i.e., non-negative values summing to 1), denoted as $\boldsymbol{p}_i = \frac{e^{w_i}}{\sum_j e^{w_j}}$ (see Footnote on page 109). The neuron's activation is then represented as the weighted sum of the

outputs from all 16 logic functions, as expressed in equation (8.2), where $f_i$ represents the $i^{th}$ real-valued operator as shown in Table 8.1 below, with $q_1$ and $q_2$ serving as the inputs to the neuron.

$$a' = \sum_{i=0}^{15} \boldsymbol{p}_i f_i(q_1, q_2) = \sum_{i=0}^{15} \frac{e^{\boldsymbol{w}_i}}{\sum_j e^{\boldsymbol{w}_j}} f_i(q_1, q_2) \tag{8.2}$$

**Table 8.1:** List of all real-valued binary logic operations.

| ID | Operator | Real-valued | 00 | 01 | 10 | 11 |
|---|---|---|---|---|---|---|
| 0 | False | 0 | 0 | 0 | 0 | 0 |
| 1 | $x \wedge y$ | $x.y$ | 0 | 0 | 0 | 1 |
| 2 | $\overline{x \Rightarrow y}$ | $x - xy$ | 0 | 0 | 1 | 0 |
| 3 | $x$ | $x$ | 0 | 0 | 1 | 1 |
| 4 | $\overline{x \Leftarrow y}$ | $y - xy$ | 0 | 1 | 0 | 0 |
| 5 | $y$ | $y$ | 0 | 1 | 0 | 1 |
| 6 | $x \oplus y$ | $x + y - 2xy$ | 0 | 1 | 1 | 0 |
| 7 | $x \vee y$ | $x + y - xy$ | 0 | 1 | 1 | 1 |
| 8 | $\overline{x \vee y}$ | $1 - (x + y - xy)$ | 1 | 0 | 0 | 0 |
| 9 | $\overline{x \oplus y}$ | $1 - (x + y - 2xy)$ | 1 | 0 | 0 | 1 |
| 10 | $\bar{y}$ | $1 - y$ | 1 | 0 | 1 | 0 |
| 11 | $x \Leftarrow y$ | $1 - y + xy$ | 1 | 0 | 1 | 1 |
| 12 | $\bar{x}$ | $1 - x$ | 1 | 1 | 0 | 0 |
| 13 | $x \Rightarrow y$ | $1 - x + xy$ | 1 | 1 | 0 | 1 |
| 14 | $\overline{x \wedge y}$ | $1 - xy$ | 1 | 1 | 1 | 0 |
| 15 | True | 1 | 1 | 1 | 1 | 1 |

Adapted from: Petersen *et al.*, 2023.

In a classification setting, where multiple neurons contribute to each class output, the output neurons are then grouped and aggregated using bit-counting to determine *class scores.* For this, the authors of the article chose an aggregation formula (8.3) that accounts for a heuristic determined parameter, $\tau$, which controls the normalization, and an optional offset, $\beta$

(Petersen *et al.*, 2023). This aggregation formula sums the probabilities within each group of neurons and the results are transformed into probabilities via *softmax cross-entropy loss*[25].

$$\hat{y} = \sum_{j=\frac{in}{k}+1}^{(i+1)n} \frac{a_j}{\tau} + \beta \tag{8.3}$$

After training, the output of each neuron is *discretized* by selecting the most likely logic function for each neuron, allowing fast inference using only Boolean operations. This discretization introduces minimal accuracy loss (Petersen *et al.*, 2023). For regression tasks, $\tau$ and $\beta$ help adjust the output range, enabling predictions beyond the [0, 1] interval (Petersen *et al.*, 2023).

## 8.4 Methodology

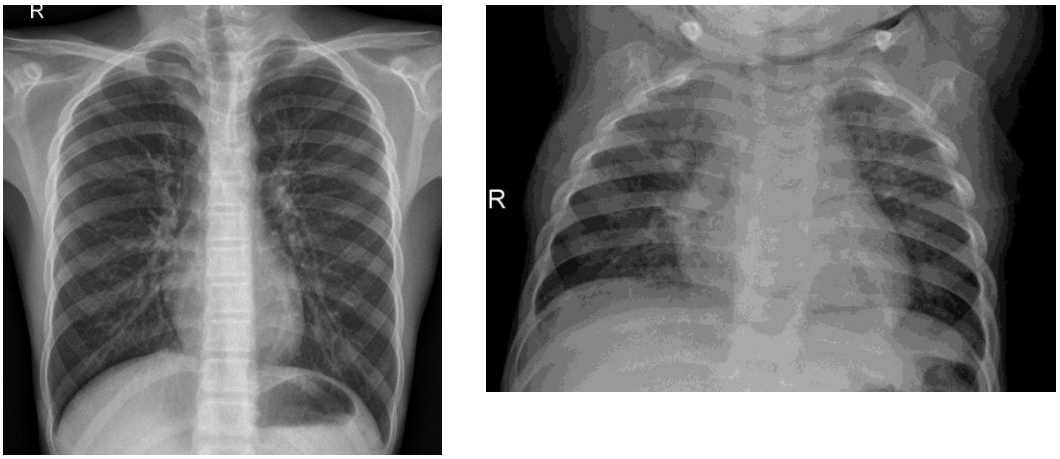
The methodology for this part of the dissertation involved a series of steps, such as analyzing the relevant literature, preparing data, training both NNs models, and analyzing the results.

The initial step consisted of literature review and exploration of the provided resources. This consisted of a thorough reading and analysis of the primary research article *Deep Differentiable Logic Gate Networks*, by Petersen *et al.* (2023), with a focus on understanding the models and tools presented by the authors. The resources provided by the authors, including any code or packages, were explored and analyzed in detail to gain insights into their implementation of the LGNs and to understand the specific parameters and techniques involved.

[25] Cross-entropy loss function is a way to measure how well a classification model's predictions match the actual class labels. It's commonly used in ML for training models on classification tasks, especially when the model outputs probabilities for different classes, like after a softmax layer. For a single example, if the true class label is $y$ and the model's predicted probability for that class is $p(y)$, the cross-entropy loss is $-\log(p(y))$, i.e., the loss is low when $p(y)$ (the model's probability for the correct class) is close to 1, and high when $p(y)$ is close to 0.

This was followed by the dataset selection, which was a key part of this study and involved identifying a suitable and challenging dataset. The dataset selection process emphasized finding data that would adequately test the capabilities of both the LGN and FFN models, providing a meaningful basis for analysis. For this, a ready-to-implement image classification dataset with an appropriate size for a NN model was searched. The dataset selected was a Pneumonia dataset, an image classification dataset with greyscale images. An example of the images composing the dataset is shown in Figure 8.2.

**Figure 8.2:** On the left side a normal case and on the right side a lung opacity case. Both examples from the test set.




Source: Breviglieri, P.C., 2019. Retrieved from https://www.kaggle.com/datasets/pcbreviglieri/pneumonia-xray-images/data.

The original dataset was undersampled to match balanced classes criterium, preventing skewing the model's training and results. The results of this process are represented on Table 8.2 below.

Additionally, as seen in Figure 8.2, the images composing this dataset vary in size. To address this, all images were rescaled to a predefined size of $40 \times 40$ pixels using the *TorchVision* package. This size was chosen after experimenting with different resolutions, as it provided the best balance for the model's performance. Larger image sizes were avoided because they would result in an input vector that grows quadratically, requiring a significantly larger network to accommodate the increased input dimensions, which was impractical due to

memory limitations on the available computational resources. Conversely, smaller image sizes led to a noticeable drop in model performance, making $40 \times 40$ pixels the optimal choice.

**Table 8.2:** Number of observations after the elimination process.

| | Normal Cases | Lung Opacity Cases | Total of Observations | Percentage |
|---|---|---|---|---|
| **Training Set** | 1267 | 1267 | 2534 | 80% |
| **Validation Set** | 158 | 158 | 316 | 10% |
| **Testing Set** | 158 | 158 | 316 | 10% |
| **Total** | 1583 | 1583 | 3166 | 100% |

## 8.5 Training an LGN

The authors of the article, *Deep Differentiable Logic Gate Networks* (2023), have published a GitHub repository which, alongside the accompanying article, includes *PyTorch* packages containing the essential scripts required to implement an LGN. These resources were used as a starting point for the experiments on this dissertation.

Recovering Figure 8.1 above, the image pixels were transformed to a greyscale into values on the interval $[0,1]$. However, the LGN only accepts binary inputs, so every greyscale value, i.e., every pixel, was discretized and transformed in a 1-dimensional vector with $n$ level-entries. For the purposes of these experimentations, it was defined "$n_{levels}$" $= 7$. For example, if one pixel transformed to a greyscale has a value of 0,71 then this pixel would be represented as (8.4)[26] below.

[26] Given the value of 0,71 and a tensor with $n_{levels} = 7$, then $0{,}71 * 7 = 4{,}97 \approx 5$, which means it would be represented as a 1-dimensional tensor with 0's in every position except for the $5^{th}$ position.

$$pixel_x = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \rightarrow \boxed{5^{th}\ Position} \tag{8.4}$$

This transformation occurs for every pixel on the selected dataset. This means that every image in the dataset will be represented as $n_{levels} \times image_{size} \times image_{size}$ tensor[27]. However, this would be a cube and LGNs are not prepared to receive a 3D format as inputs. For this reason, the first layer of the LGN is a so-called *Flatten* layer. This transforms the 3-dimensional tensor that represents each image into a 1-dimensional tensor of size $n_{levels} * image_{size} * image_{size}$, where the $n_{levels} = 7$ and the $image_{size} = 40$. This means that every image on the dataset entered the network as a 1-dimensional binary tensor of size $11200$.

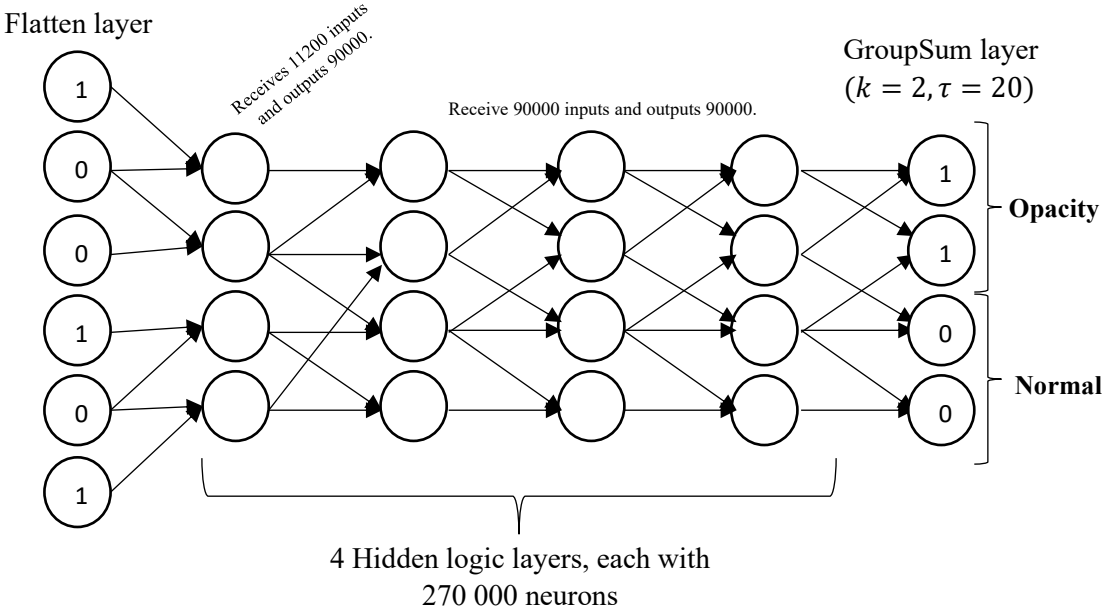
These inputs are then processed by several layers of neurons. After an initial random initialization of the connections between neurons, these connections remain fixed. Each neuron is represented by a probability distribution over possible logic functions, which is learned during training. After training, the most likely logic function for each neuron is selected and used at inference for the classification into either "Opacity" or "Normal". The network created for this experimentation is represented in Figure 8.3.

Multiple neurons contribute to the output for each class, and their results are combined using bit-counting to generate class scores. The bit-counting sum is implemented through a Python function, the *GrupSum* function, according to the aggregation formula in (8.3) above, where $\beta$ is already defined by the original architecture and $\tau = 20$ is defined heuristically.

[27]In *PyTorch*, a tensor is a mathematical object similar to a multi-dimensional array or matrix, used to store and manipulate numerical data. Tensors can be thought of as generalizations of scalars, vectors (1D arrays), and matrices (2D arrays) to higher dimensions. They are designed to handle complex mathematical operations efficiently and can take advantage of powerful hardware like GPUs for faster computations. Unlike traditional mathematical arrays, *PyTorch* tensors are also equipped with tools to automatically calculate derivatives, which is essential for optimizing functions in ML.

In an LGN, the final layer before computing the loss is typically the *GrupSum layer*, which aggregates the outputs of the differentiable logic gates into a vector of real-valued scores (one per class). These are then passed through a *softmax function*, which converts them into a probability distribution over the possible classes.

**Figure 8.3:** The LGN.

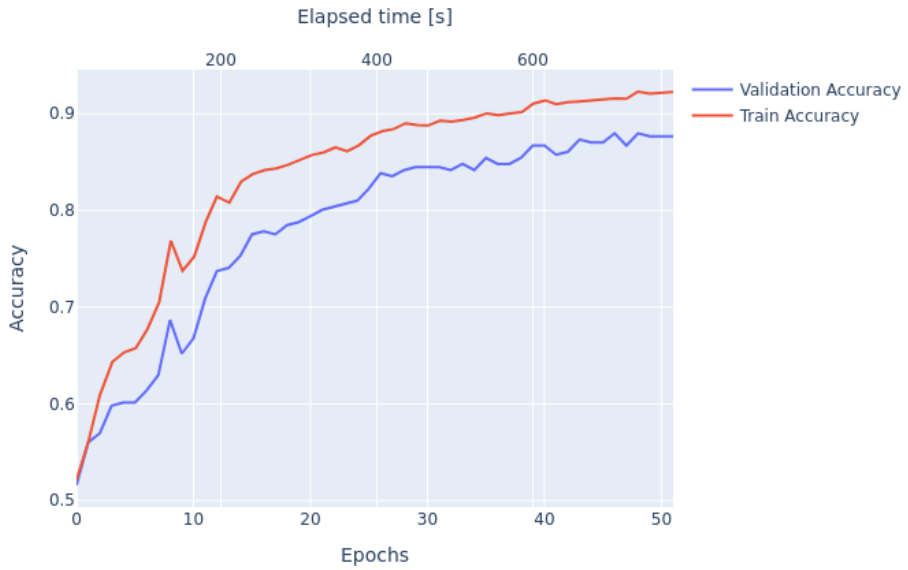



The resulting probabilities are compared to the true labels using the *cross-entropy loss function*, which quantifies the difference between the predicted distribution and the actual class label. Because all components are differentiable, the model can be trained end-to-end using stochastic gradient descent optimized with Adam through standard backpropagation.

After several experiments on this dataset, a batch size of 4 and a learning rate of $0.0001$ were defined. An Early-Stopping technique was applied to the training with a "patience" of 4. This is used to prevent overfitting and save computational resources by stopping the training process early if the performance of the validation set does not improve for 4 consecutive epochs.

The results of the LGN model are shown in Figure 8.4. As shown, the LGN model was stopped during training by the predefined "patience" at epoch 46, with an $accuracy_{test} = 89.9\%$. It took 705 seconds to get to this epoch.

**Figure 8.4:** Evolution of the train and validation accuracies during train of the LGN.




## 8.6 Training an FFN

Unlike LGNs, which are restricted to binary inputs and require data to be transformed into a 1-dimensional tensor representation, FFNs are not bound by this limitation. FFNs can process numerical inputs in their original form, allowing the resized images in the selected dataset to be fed directly into the network as 1-dimensional floating-point tensors. This flexibility simplifies the preprocessing pipeline enabling the FFN to work directly with a flattened version of the resized images pixels tensors.

As explained in Chapter 7, in an FFN, each layer applies a linear transformation (fully connected layer) followed by an activation function. In this case, the ReLU activation function was used. The initial weights of the network were initialized by drawing from a standard normal distribution.

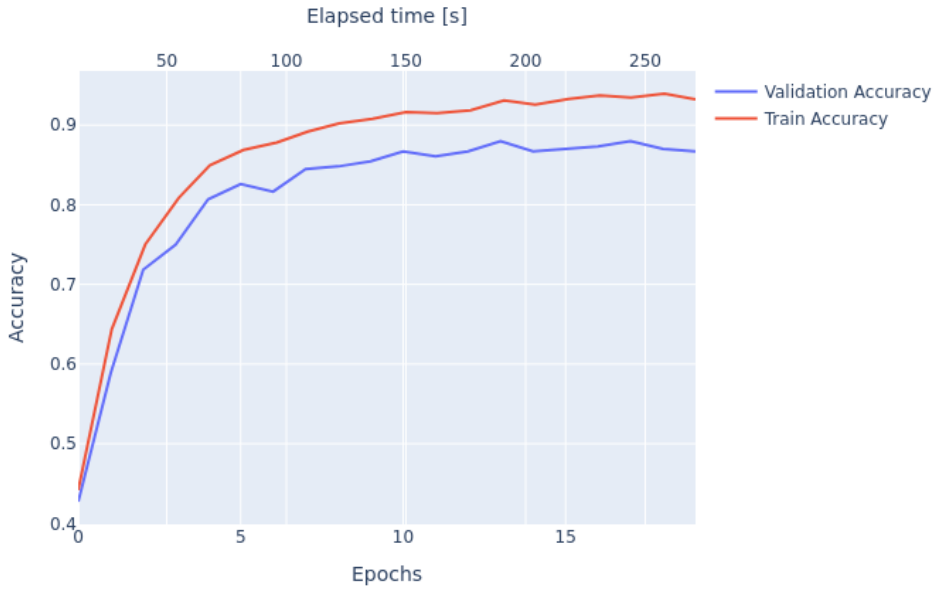
During training, the network adjusts the weights of the linear layers to minimize a loss function, the cross-entropy loss function. This loss function is minimized through a stochastic gradient descent algorithm optimized with Adam, with a batch size of 32. After several experiments on this dataset, a learning rate of 0.0001 was defined. Like for the LGN model,

an Early-Stopping technique will also be applied for the FFN training, in this case with a "patience" of 5.

The FFN created for this experiment is composed of one first so-called *Flatten layer* (that works in the same way as the *Flatten* layer on the LGN model), has a total of 4 hidden layers composed of 1 150 neurons each, and, finally, an output layer that takes 1 150 inputs and outputs the classes (0 or 1).

Figure 8.5. shows the results of the FFN. The FFN was stopped during training by the predefined "patience" at epoch 13, with an $accuracy_{test} = 80.4\%$. It took 191 seconds to get to this epoch.

**Figure 8.5:** Evolution of the train and validation accuracies during train of the FFN.




Due to time constraints, it was not possible to perform a comprehensive hyperparameter search[28], which is typically essential to identify the optimal configuration for the models. As a result, the selected hyperparameters were based on standard values or

[28] Hyperparameter search refers to the process of systematically exploring and optimizing the settings (hyperparameters) of a NN that govern its training and structure. Examples of hyperparameters include the learning rate, batch size, number of layers, and number of neurons in each layer. Unlike the weights, which are learned during training, hyperparameters are predefined and can significantly impact the model's performance. The search process may involve techniques like grid search, random search, or more advanced methods such as Bayesian optimization.

minimal tuning and may not represent the best possible settings for achieving peak performance. Consequently, while the models demonstrate the intended functionality and provide useful insights, their performance cannot be guaranteed to reflect the highest achievable results under different parameter configurations.

## 8.7 Resources

Both models were trained in *PyTorch* using a NVIDIA RTX 2070 GPU on an Intel Xeon® E5-2695 v2 CPU.

Graphics Processing Units (GPUs), like NVIDIA RTX 2070 used for this dissertation, are specialized hardware created to efficiently perform mathematical operations that allow for graphics processing, such as matrix operations. These operations are also important to perform gradient-based optimizations in NN training, making these devices also ideal for these tasks and significantly reducing the time required to train complex models.

The complete list of resources and correspondent versions can be consulted on Table 8.3. The script is available in Appendix X.

**Table 8.3:** Versions of the software and packages used.

| Software / Package | Version |
|---|---|
| Operating System | Ubuntu 20.04 |
| Python | 3.8.10 |
| Torch | 1.9.0 |
| TorchVision | 0.10.0 |
| CUDA | 11.1.1 |
| JupyterLab | 4.2.5 |
| Pandas | 1.5.3 |
| NumPy | 1.24.4 |
| Plotly | 5.24.1 |
| Difflogic[29] | Git Version on September 2024 |

[29] Available at: https://github.com/Felix-Petersen/difflogic

## 8.8 Discussion

The results of training two different models were presented. The LGN model demonstrated a training time of 701 seconds, achieving an accuracy of 89.9%. This result indicates that the LGN was able to learn the underlying patterns in the dataset effectively, producing a high level of accuracy. In contrast, the FFN model required a shorter training time of 191 seconds but achieved a lower accuracy of 80.4%. While the FFN was able to learn from the data in less time, its performance was not as strong as that of the LGN.

These results suggest that while the training process of the LGN is more computationally intensive, this model is more capable of capturing complex relationships within the data, as evidenced by its higher accuracy. On the other hand, the FFN model, with its faster training time, may be more suitable for scenarios where speed in training is a priority, though it sacrifices some predictive accuracy.

Although both models were trained with a similar number of total trainable parameters (5 760 000 and 5 814 402 for the LGN and FFN, respectively), the FFN obtained is not guaranteed to be the most performant for this dataset.

Due to the lack of detailed information in the original article regarding the measurement of inference time, it is not possible to provide any insights or conclusions on this aspect.

## 8.9 Conclusion

In this chapter, the design and implementation of LGNs were explored, including their differentiable variants, and their performance were assessed through experimentation. The methodology for training and evaluating LGNs was discussed, along with the unique challenges they present. The experiments executed demonstrated that LGNs can be trained efficiently while maintaining high levels of accuracy.

LGNs offers significant advantages in terms of computational and memory efficiency. By relying solely on bitwise operations instead of floating-point arithmetic, LGNs seem to be

ideal for real-time applications and resource-constrained environments such as embedded systems and low-power devices (Petersen *et al.*, 2023). Furthermore, the absence of weight matrices eliminates the memory overhead typically associated with traditional NNs, resulting in a smaller memory footprint. The combination of binary inputs and sparse architecture allows LGNs to be much more memory-efficient than dense neural networks, making them a highly suitable solution for applications where both computational and storage resources are limited (Petersen *et al.*, 2023).

Additionally, the chapter addressed how LGNs can be trained, and it presents a practical example that shows a possible implementation of this specific network, as well as the used software environment. The results suggest that LGNs have the potential to be powerful tools, having achieved high levels of accuracy in our experiment.

In summary, the experiments in this chapter underline the strengths and limitations of LGNs. Future work could explore further optimizations in model architecture applying for example hyperparameter search and regularization techniques.

# CONCLUSION

In the ever-evolving landscape of modern computing, the intersection of numerical systems and Boolean algebra remains a foundational pillar, influencing the development of both hardware and software systems. From the basic principles of binary, decimal, octal, and hexadecimal number systems, to the more advanced applications of Boolean algebra in digital circuits, these mathematical tools are indispensable in shaping how information is processed and computed. Of particular note is the binary system, which, due to its compatibility with electronic devices, underpins the core of digital computation.

Boolean algebra, which began as an abstract theoretical framework by George Boole, has since evolved into a cornerstone of technological advancement. Its logical operations, such as AND, OR, and NOT, along with principles like De Morgan's laws and the duality principle, provide essential methods for simplifying and manipulating logical expressions. This has made it possible to construct complex, yet efficient, computational systems.

The historical progression from abstract theory to tangible, real-world applications underscore the enduring relevance of Boolean algebra in technology today. It is through this integration that the link between mathematical logic and technological progress is most clearly seen, providing the foundation for future breakthroughs in computing. Boolean algebra's influence is not limited to basic circuits; it extends to advanced digital systems, where its application in combinational and sequential circuits offers solutions to a wide array of computational needs.

Combinational circuits, which operate based on the current state of inputs, and sequential circuits, which incorporate memory to track previous states, are both critical to digital logic. These circuits are the building blocks of all modern electronics, from simple arithmetic operations to more complex systems like flip-flops and latches, which enable memory storage and synchronization in digital systems. The evolution of these circuits exemplifies the practical applications of Boolean logic in solving both simple and intricate computational problems.

As technology advances, so does the integration of Boolean algebra into more sophisticated domains like artificial intelligence (AI). The development of LGNs, as proposed by Petersen *et al.* (2023), demonstrates how Boolean logic can be applied in NN architectures. These networks leverage logical gates to perform computations more efficiently, particularly in scenarios where computational resources are limited. Through the use of differentiable relaxation techniques, LGNs bridge the gap between traditional Boolean logic and modern ML, offering a new approach to network design that combines logical precision with computational efficiency.

In practical terms, LGNs have proven effective in tasks like image classification. The ability of LGNs to operate with binary inputs and outputs, combined with their low memory footprint, makes them ideal for real-time applications and embedded systems, where resource constraints are a critical consideration. These networks, while requiring longer training times, demonstrate a high predictive accuracy which is beneficial in resource-constrained environments.

Looking ahead, further optimization of LGNs through hyperparameter tuning and advanced regularization techniques may unlock even greater performance and generalization capabilities. The potential for LGNs in fields such as medical diagnostics, embedded systems, and IoT devices is immense, offering a glimpse into a future where computational efficiency and accuracy are harmoniously balanced. This research underscores the importance of exploring alternative NN architectures that prioritize efficiency without sacrificing performance, ultimately contributing to the advancement of AI and computational technology.

In conclusion, the journey from Boolean algebra's theoretical origins to its application in modern digital circuits and advanced NNs reflects a remarkable fusion of mathematics and technology. Boolean logic remains an integral force in shaping the future of computing, providing the foundational principles upon which innovative technologies are built. Whether through the construction of efficient digital circuits or the development of advanced AI systems like LGNs, Boolean algebra continues to play a pivotal role in the ongoing evolution of computational systems. The exploration of such alternatives demonstrates the boundless

possibilities that arise when traditional mathematical principles are applied in novel and cutting-edge ways.

# REFERENCES


Aggarwal, C. (2018). *Neural Networks and Deep Learning: A Textbook.* New York: Springer International Publishing AG. doi:10.1007/978-3-319-94463-0

Arroz, G., Monteiro, J., & Oliveira, A. (2019). *Computer Architecture: Digital Circuits to Microprocessors.* World Scientific.

Breviglieri, P. C. (2019). Pneumonia X-ray Images Dataset [Adapted from Paul Mooney's 'Chest X-Ray Images (Pneumonia)' dataset]. Kaggle. Retrieved from https://www.kaggle.com/datasets/pcbreviglieri/pneumonia-xray-images/data.

Brunton, S., & Kutz, J. (2019). Neural Networks and Deep Learning. In S. L. Brunton, & J. N. Kutz, *Data Driven Science & Engineering: Machine Learning, Dynamical Systems, and Control* (pp. 227-291). Cambridge University Press.

Dr. Bhagat, A. (n.d.). *Digital Circuits And Logic Design.* Punjab: LP University. Retrieved January 2024, from http://eslm.lpude.in/computer_application/ad/DCAP108_DIGITAL_CIRCUITS_AND_LOGIC_DESIGNS/index.html#p=2

Goodfellow I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press. Retrieved Mai 2024, from http://www.deeplearningbook.org

Mano, M. M., & Ciletti, M. (2013). *Digital Design: With An introduction to the Verilog HDL* (5 ed.). Pearson.

Mano, M. M., & Kime, C. (2014). *Logic and Computer Design Fundamentals* (fourth ed.). Pearson New International Edition.

Nielsen, M. (2015). *Neural Networks and Deep Learning.* Determination Press. Retrieved January, 2024, from http://neuralnetworksanddeeplearning.com

Petersen, F., Borgelt, C., Kuehne, H., & Deussen, O. (2023). Deep differentiable logic gate networks. *Advances in Neural Information Processing Systems, 35*, pp. 2006-2018.

Qin, H., Gong, R., Liu, X., Bai, X., Song, J., & Sebe, N. (2020). Binary neural networks: A survey. *Pattern Recognition, 105*, p. 107281.

T.P., S. (2018). *Boolean Algebra.* Lulu.com.

Yang, D. K.-p. (2020). Boolean Algebra and Digital logic. In K.-p. Dr. Yang, *CMPS 375 - Computer Architecture* (pp. 137-209). Louisiana: Southeastern Louisiana University. Retrieved December 2023, from https://pt.scribd.com/presentation/460832226/CMPS375ClassNotesChap03-pptx

# APPENDIX I: OCTAL AND HEXADECIMAL SYSTEMS

## I. Octal and Hexadecimal Systems

The octal and the hexadecimal systems are probably the least common in everyday life. However, that does not mean that they are of less importance. In fact, both are extremely important in the world of computer systems and engineering.

As shown, the use of binary representation is preferred when implementing digital systems (Mano & Kime, 2014). Nevertheless, the octal and hexadecimal systems play an important role as well.

The octal number system, also known as the base (or *radix*) 8 system, uses eight symbols (0,1,2,3,4,5,6,7) and it is occasionally used in computing, particularly in contexts where representing binary values in a more compact form is advantageous. Taking as an example the number 151.4. In (I.1) its equivalent decimal value is determined, expanding the number in a power series of base 8.

$$(151.4)_8 = (1 * 8^2) + (5 * 8^1) + (1 * 8^0) + (4 * 8^{-1}) = (105.5)_{10} \qquad (I.1)$$

Each octal digit corresponds to a unique combination of three binary digits. For example, the octal number 155, reads 001 101 101 in binary, as $(1)_8 = (001)_2$ and $(5)_8 = (101)_2$. This grouping simplifies the representation of binary numbers and facilitates conversions between the two numeral systems. For this reason, this number system is commonly used in the world of computer systems and engineering.

For example, the Unix/Linux system uses octal notation to set file permissions, with each octal digit representing a set of permissions (read, write, execute) for the owner, group, and other users. For example, the command "`chmod 755 file`" assigns permissions where "7" grants all permissions (read, write, execute) to the owner, while "5" allows read and execute permissions for both the group and other users.

Hexadecimal, also known as the base (or *radix*) 16 system, is favored because each digit represents four binary digits, aligning well with the common grouping of 8, 16, and 32

bits[30] in computer architectures. It is often employed to represent binary-coded values in a more concise and human-friendly way.

This system employs sixteen symbols, including the digits (0,1,2,3,4,5,6,7,8,9), followed by the 6 letters (A, B, C, D, E, F) to represent values between 10 and 15. Taking the hexadecimal number $D45A$, in (I.2) its equivalent decimal value is shown.

$$(D45A)_{16} = (13 * 16^3) + (4 * 16^2) + (5 * 16^1) + (10 * 16^0) = (54\,362)_{10} \qquad (I.2)$$

Like the octal system, the conversion between the binary system and the hexadecimal systems is also very simple. To convert a binary number to hexadecimal value one should divide the binary number into four groups (instead of three in the octal systems). Taking the previous hexadecimal number as an example its binary equivalent is 1101 0100 0101 1010, because $(D)_{16} = (1101)_2$, $(4)_{16} = (0100)_2$, $(5)_{16} = (0101)_2$ and $(A)_{16} = (1010)_2$.

For example, in HTML, CSS, and graphic design, colors are often expressed in hexadecimal, with each pair of digits representing red, green, and blue (RGB) values ranging from 0 to 255. For instance, the hex code #FF5733 represents an orange color, where FF is the maximum value for red, 57 is the green component, and 33 is the blue component.

Both systems allow for a representation of binary in fewer digits and, because the base of both systems are powers of 2 (8 and 16 respectively), the conversion to binary is more direct. For this reason, they play an important role in computer sciences and digital circuits. Hexadecimal is often preferred as it aligns neatly with bytes, making data easier to read and work with in computing.

[30] In computing and digital communications, a *bit* is the basic unit of information. The term "bit" is a contraction of "binary digit," and it represents a state in a binary system. A bit can have only one of two values, typically represented as 0 or 1, false or true, off or on. These two values correspond to the binary numeral system. Bits are used to represent and store data in computers, and they are the smallest unit of data in a computer's memory.

# APPENDIX II: BOOLEAN ALGEBRA AND SET THEORY

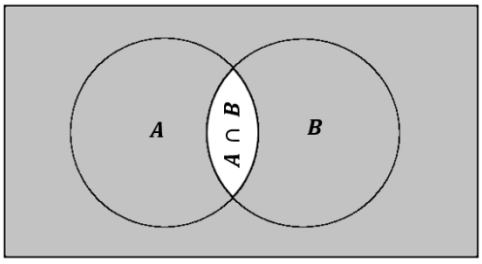
## II. Boolean Algebra and Set Theory

To grasp the mathematical underpinnings of Boolean algebra, it is essential to explore its formal representation and operations. The relationship between Boolean algebra and set theory is profound, with Boolean expressions finding equivalence in set operations.

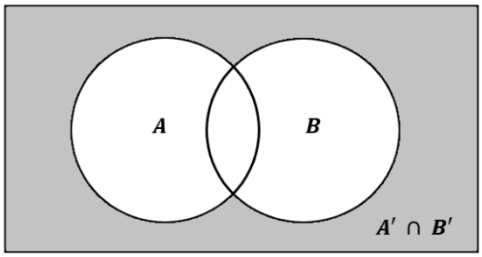
Boolean algebra is grounded in set theory and provides a mathematical framework for working with binary variables and logical operations. As previously seen, the axioms, laws, and operations of Boolean algebra define how logical operations can be manipulated. Extending their utility to set theory, De Morgan's laws provide a valuable rule for determining the complement of the union or intersection of sets.

The complement of the intersection of sets A and B equals the union of their complements as shown in Figure II.1 below, while the complement of the union of sets A and B equals the intersection of their complements as seen also in Figure II.1 (Bhagat).

**Figure II.1:** Complement of the intersection or union of sets.




$(A \cap B)' = A' \cup B'$



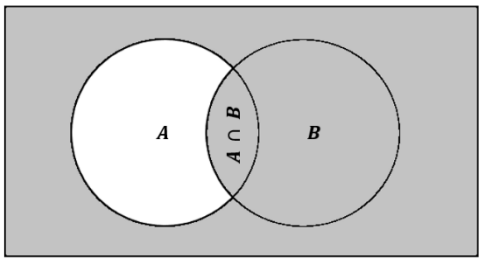

$(A \cup B)' = A' \cap B'$

Additionally, these laws apply to set differences. For sets $A$ and $B$, the complement of the difference $A - B$ equals the union of the complement of $A'$ and set $B$, as in Figure II.2 below, i.e., the elements not in the set difference between $A$ and $B$ are the same as the elements not in $A$, combined with the elements in $B$ (Bhagat).

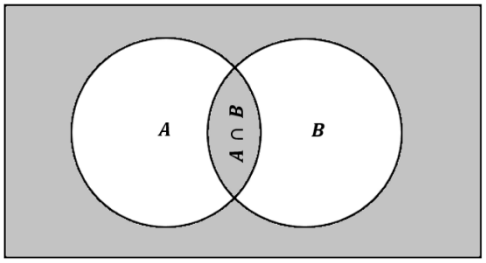
Similarly, the complement of the symmetric difference of sets $A$ and $B$, $A \Delta B$, is equal to the intersection of the union of the complement of $A$ and set $B$, and the union of set $A$ and the complement of $B$, $(A - B) \cup (B - A)$, as also represented in Figure II.2, i.e., the elements

not in the symmetric difference of $A$ and $B$, are the elements that are either in both $A$ and $B$ or in neither $A$ nor $B$ (Bhagat).

**Figure II.2:** Set differences.




$(A - B)' = A' \cup B$




$(A \Delta B)' = (A' \cup B) \cap (A \cup B')$

De Morgan's laws play a crucial role in proof techniques, enabling mathematicians and logicians to establish equalities between diverse logical expressions.

# APPENDIX III: KARNAUGH'S MINIMIZATION

## III. Karnaugh's Minimization for a Four-Variable Function

| $x$ | $y$ | $z$ | $w$ | | $f$ | Minterms | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 | $\boldsymbol{m_0}$ | $\boldsymbol{\bar{x}\,\bar{y}\,\bar{z}\,\bar{w}}$ |
| 0 | 0 | 0 | 1 | | 1 | $\boldsymbol{m_1}$ | $\boldsymbol{\bar{x}\,\bar{y}\bar{z}w}$ |
| 0 | 0 | 1 | 0 | | 1 | $\boldsymbol{m_2}$ | $\boldsymbol{\bar{x}\,\bar{y}z\bar{w}}$ |
| 0 | 0 | 1 | 1 | | 0 | $m_3$ | $\bar{x}\bar{y}zw$ |
| 0 | 1 | 0 | 0 | | 0 | $m_4$ | $\bar{x}y\bar{z}\bar{w}$ |
| 0 | 1 | 0 | 1 | | 0 | $m_5$ | $\bar{x}y\bar{z}w$ |
| 0 | 1 | 1 | 0 | | 1 | $\boldsymbol{m_6}$ | $\boldsymbol{\bar{x}yz\bar{w}}$ |
| 0 | 1 | 1 | 1 | | 0 | $m_7$ | $\bar{x}yzw$ |
| 1 | 0 | 0 | 0 | | 1 | $\boldsymbol{m_8}$ | $\boldsymbol{x\bar{y}\bar{z}\bar{w}}$ |
| 1 | 0 | 0 | 1 | | 1 | $\boldsymbol{m_9}$ | $\boldsymbol{x\bar{y}\bar{z}w}$ |
| 1 | 0 | 1 | 0 | | 1 | $\boldsymbol{m_{10}}$ | $\boldsymbol{x\bar{y}z\bar{w}}$ |
| 1 | 0 | 1 | 1 | | 0 | $m_{11}$ | $x\bar{y}zw$ |
| 1 | 1 | 0 | 0 | | 0 | $m_{12}$ | $xy\bar{z}\bar{w}$ |
| 1 | 1 | 0 | 1 | | 0 | $m_{13}$ | $xy\bar{z}w$ |
| 1 | 1 | 1 | 0 | | 0 | $m_{14}$ | $xyz\bar{w}$ |
| 1 | 1 | 1 | 1 | | 0 | $m_{15}$ | $xyzw$ |

**1.**We start by representing our function, $f$, in a truth table. In circuit design its common practice starting the process writing a truth table that illustrates our intentions.

With this table we could write $f$ in its disjunctive normal form:

$$f = \bar{x}\bar{y}\bar{z}\bar{w} + \bar{x}\bar{y}\bar{z}w + \bar{x}\bar{y}z\bar{w} + \bar{x}yz\bar{w} + x\bar{y}\bar{z}\bar{w} + x\bar{y}\bar{z}w + x\bar{y}z\bar{w}$$

However, this expression isn't optimal, as its implementation would require to many logic gates and would be expensive. This expression can be optimized through Karnaugh's minimization.

**2.** Given a truth table, **Karnaugh's minimization** starts with the deduction of the K-map.

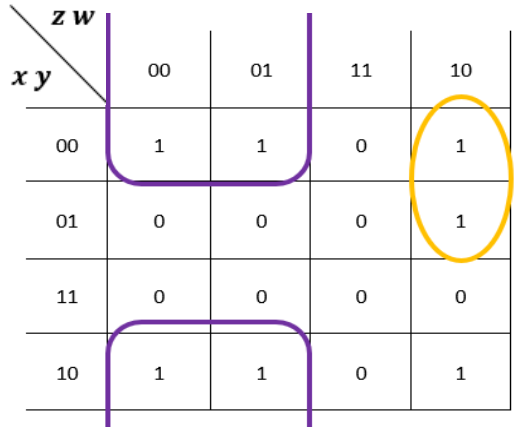
| $xy$ \ $zw$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 1 |
| 01 | 0 | 0 | 0 | 1 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 1 | 1 | 0 | 1 |

| $xy$ \ $zw$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $\boldsymbol{\bar{x}\,\bar{y}\,\bar{z}\,\bar{w}}$ | $\boldsymbol{\bar{x}\,\bar{y}\bar{z}w}$ | $\bar{x}\bar{y}zw$ | $\boldsymbol{\bar{x}\,\bar{y}z\bar{w}}$ |
| 01 | $\bar{x}y\bar{z}\bar{w}$ | $\bar{x}y\bar{z}w$ | $\bar{x}yzw$ | $\boldsymbol{\bar{x}yz\bar{w}}$ |
| 11 | $xy\bar{z}\bar{w}$ | $xy\bar{z}w$ | $xyzw$ | $xyz\bar{w}$ |
| 10 | $\boldsymbol{x\bar{y}\bar{z}\bar{w}}$ | $\boldsymbol{x\bar{y}\bar{z}w}$ | $x\bar{y}zw$ | $\boldsymbol{x\bar{y}z\bar{w}}$ |

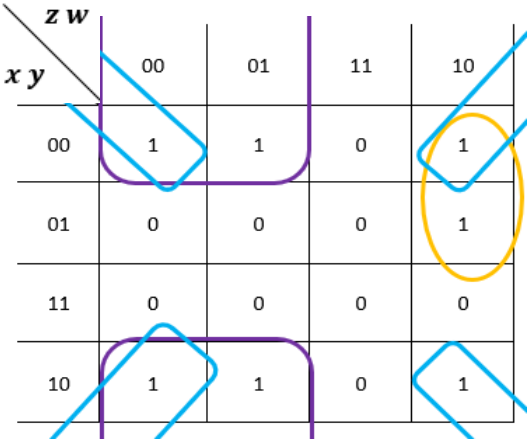
→ **3.** The adjacent 1's are looped.

| *x y* \ *z w* | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 1 |
| 01 | 0 | 0 | 0 | 1 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 1 | 1 | 0 | 1 |

**A.** We start looping the pair $m_2, m_6$, which yields the term: $\bar{x}z\bar{w}$, since $\bar{x}\bar{y}z\bar{w} + \bar{x}yz\bar{w} = \bar{x}(\bar{y} + y)z\bar{w} = \bar{x}z\bar{w}$




**B.** Then, we loop the quad $m_0, m_1, m_8, m_9$, which yields the term: $\bar{y}\,\bar{z}$, since $\bar{x}\,\bar{y}\,\bar{z}\,\bar{w} + \bar{x}\,\bar{y}z\bar{w} + x\bar{y}\bar{z}\bar{w} + x\bar{y}\bar{z}w = \bar{y}\bar{z}(2\bar{x} + 2x + 2w + 2\bar{w} = \bar{y}\bar{z}$.




**C.** Finally, we wrap the corners into a quad $m_0, m_2, m_8, m_{10}$, which yields the term: $\bar{y}\,\bar{w}$, since $\bar{x}\,\bar{y}\,\bar{z}\,\bar{w} + \bar{x}\bar{y}z\bar{w} + x\bar{y}\bar{z}\bar{w} + x\bar{y}z\bar{w} = \bar{y}\,\bar{w}(2\bar{x} + 2x + 2z + 2\bar{z}) = \bar{y}\,\bar{w}$.

**4.** We obtain the simplified function to be implemented in a digital circuit:

$$f = \bar{y}\bar{z} + \bar{y}\bar{w} + \bar{x}z\bar{w}$$

# APPENDIX IV: EQUATIONS OF THE FA

## IV. Equations of the FA

Proof of the equations for the $S$ and the $C_{out}$ of the FA presented on Figure 5.3 in Chapter 5.

- $\boldsymbol{S = \overline{C_{in}}\bar{x}y + \overline{C_{in}}x\bar{y} + C_{in}\bar{x}\bar{y} + C_{in}xy = \overline{C_{in}}(x \oplus y) + C_{in}\overline{x \oplus y} = C_{in} \oplus x \oplus y}$

- $\boldsymbol{C_{out} = xy + C_{in}x + C_{in}y = xy + C_{in}(x \oplus y)}$

    - $\boldsymbol{xy + C_{in}x + C_{in}y} = (C_{in} + \overline{C_{in}})xy + C_{in}x(y + \bar{y}) + C_{in}(x + \bar{x})y =$
      $= C_{in}xy + \overline{C_{in}}xy + C_{in}xy + C_{in}x\bar{y} + C_{in}xy + C_{in}\bar{x}y =$
      $= C_{in}xy + C_{in}xy + C_{in}xy + \bar{C}_{in}xy + C_{in}x\bar{y} + C_{in}\bar{x}y =$
      $= C_{in}xy + \bar{C}_{in}xy + C_{in}(x \oplus y) =$
      $= (C_{in} + \overline{C_{in}})xy + C_{in}(x \oplus y) =$
      $= \boldsymbol{xy + C_{in}(x \oplus y)}$

Auxiliary calculations:

- $x \oplus y = \bar{x}y + x\bar{y}$
- $\overline{x \oplus y} = \overline{\bar{x}y}.\overline{x\bar{y}} = (x + \bar{y})(\bar{x} + y) = x\bar{x} + xy + \bar{x}\bar{y} + \bar{y}\bar{y} + xy + \bar{x}\bar{y}$

# APPENDIX V: SUBTRACTION WITH THE 2'S COMPLEMENT METHOD

## V. Subtraction with the 2's Complement Method

To implement the method for subtracting two binary numbers, the initial step involves finding the 2's complement of the number that is to be subtracted from another number. To do this, first the 1's complement is determined by inverting all bits (changing 1s to 0s and 0s to 1s), and then adding 1 to the result. This addition yields the required 2's complement.

For example, to find the 2's complement of the binary number 10010, the process starts by finding its 1's complement: invert 10010, getting 01101. Adding 1 to 01101 results in the 2's complement of our original number, which is **01110**.

**Figure V.1:** Subtraction of two numbers using the 2's complement method.

$$\mathbf{11111_2 - 10010_2 = 11111_2 + 01110_2}$$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 | | | 1 | 1 | 1 | 1 | 1 | → Minuend is the first addend. |
| - | 1 | 0 | 0 | 1 | 0 | | + | 0 | 1 | 1 | 1 | 0 | → 2's complement of the subtrahend. |
| | 0 | 1 | 1 | 0 | 1 | ← | ~~1~~ | 0 | 1 | 1 | 0 | 1 | → The result is the remaining part of the sum. |
| | | | | | | | ↓ | | | | | | |
| | | | | | | | *end carry* | | | | | | → The *end carry* is discarded. |

This binary operation translates to the subtraction of the decimal numbers $31 - 18 = 13$. The binary number $11111_2 = 31_{10}$, $10010_2 = 18_{10}$ and the result $01101_2 = 13_{10}$.

Nevertheless, if the number of digits of the subtrahend and minuend differ, the process of calculating the 2's complement is slightly different, as illustrated in Figure V.2 below.

**Figure V.2:** Subtraction with different number of digits.

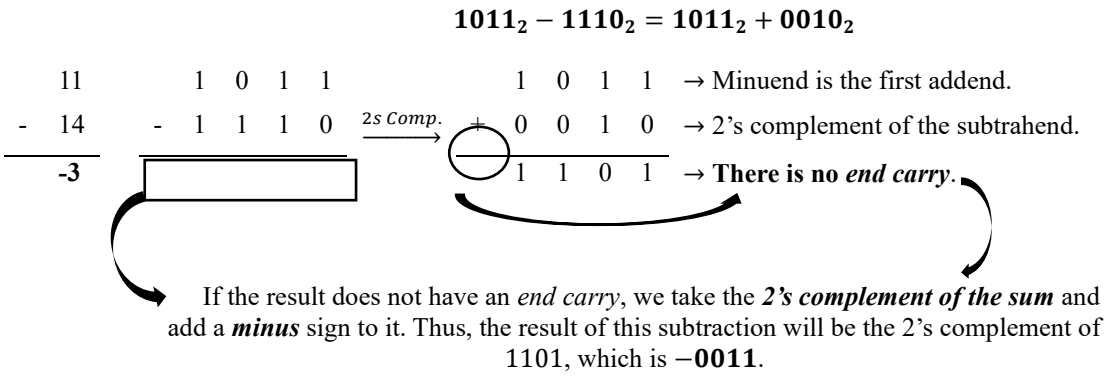
$$\mathbf{1011_2 - 110_2 = 1011_2 + 1010_2}$$

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 11 | | | 1 | 0 | 1 | 1 | | | 1 | 0 | 1 | 1 | → Minuend is the first addend. |
| - | 6 | | - | **0** | 1 | 1 | 0 | *2s Comp.* | + | 1 | 0 | 1 | 0 | → 2's complement of the subtrahend. |
| | 5 | | | 0 | 1 | 0 | 1 | | | ~~1~~ | 0 | 1 | 0 | 1 | → The *end carry* is discarded. |

If the minuend and subtrahend have different nº of digits, we **prepend zeros** to the shorter one until the nº of digits become equal.

Additionally, if the result of the subtraction does not have an *end carry*, there are also important changes, as explained in Figure V.3 below.

**Figure V.3:** Subtraction without an end carry.



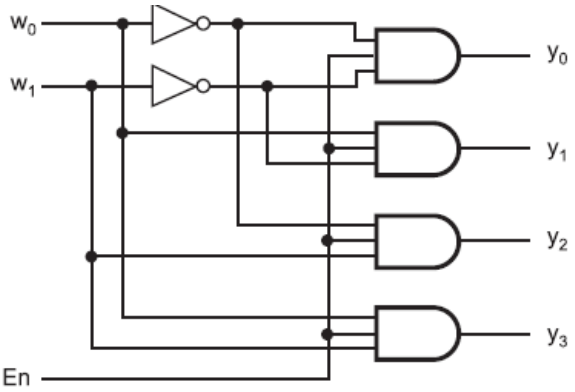
# APPENDIX VI: DECODERS AND ENCODERS

## VI. Decoders and Encoders

Decoder circuits are designed to interpret encoded information. A binary decoder is a logic circuit featuring $n$ inputs and $2^n$ outputs. At any given time, only one output is active, corresponding to a specific combination of input values. The decoder includes an enable input, $En$, which controls whether the outputs are active; if $En = 0$, all outputs are inactive. When $En = 1$, the combination of inputs $w_{n-1}, \dots w_1, w_0$ determines which output is activated. A binary code where exactly one bit is set to 1 at any moment is called *one-hot encoded*, indicating that the single bit set to 1 is considered *'hot'*.

**Figure VI.1:** Logic circuit and truth table of a 2-to-4 Decoder.

| $En$ | $w_1$ | $w_0$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | x | x[31] | 0 | 0 | 0 | 0 |




Source: Bhagat:85.

As practical example where a 3-to-8 Decoder could be used is a memory of a computer that consists of eight memory chips, each containing 8Kbytes (Mano & Ciletti, 2013; Yang, 2020). Because there are 8 chips, each with 8 Kbytes, the total memory capacity is $8 \times 8 = 64$ Kbytes. 64 Kbytes is the equivalent to approximately 65,536 addresses (Mano & Ciletti, 2013; Yang, 2020).

[31] *Don't care conditions* occur when certain input combinations of a logic function do not influence the desired outcome, allowing flexibility in assigning output values for these combinations. By treating some input combinations as "don't care," designers can simplify logic expressions and reduce the complexity of the circuit. These conditions allow for more efficient implementation of the logic circuit, potentially reducing the number of gates and components needed. These conditions are typically used to simplify the design and optimization of logic circuits. In a truth table, "don't care conditions" are usually represented by an 'X' or a '-' indicating that the output can be either 0 or 1 for those specific inputs. When minimizing Boolean expressions using K- maps, these can be grouped with either 1s or 0s to form larger groups, leading to simpler expressions.

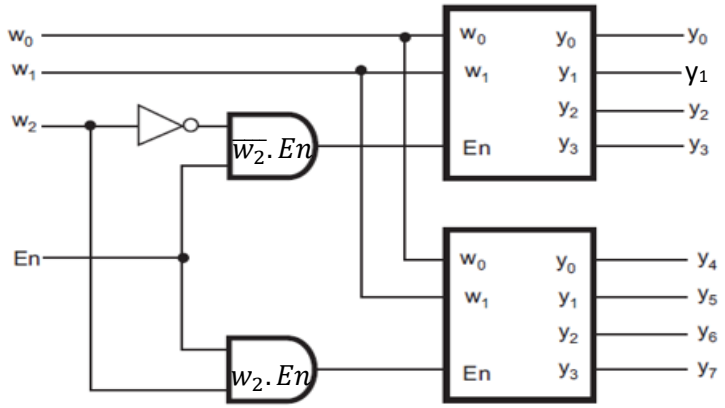
To represent each of these 65,536 addresses, 16 bits are needed (since $2^{16} = 65{,}536$). Thus, each address in this memory system is represented by a 16-bit binary number.

The leftmost 3 bits of the 16-bit address determine which of the 8 memory chips contains the specific address, i.e., addresses on chip 0 (the first chip) have the format "000X XXXX XXXX XXXX" (where X represents any bit, 0 or 1) (Yang, 2020:13). This means the addresses range from 0 to 8191. Addresses on chip 1 have the format "001X XXXX XXXX XXXX", and so on for the other chips (Yang, 2020:13).

The 3 leftmost bits of the 16-bit address are used as inputs to, for example, a 3-to-8 Decoder represented in Figure VI.2 below. This Decoder converts the 3-bit input into one of 8 outputs, where each output corresponds to one of the 8 chips. For example, if the input to the decoder is "000", it activates the output line connected to chip 0, if the input is "001", it activates the output line connected to chip 1, and so on. Only one output line of the Decoder is active at any time, ensuring that only one chip is selected based on the 3-bit input.

In this example, the 3-to-8 Decoder is essential for selecting the correct memory chip among the 8 available chips. By using the 3 high-order bits of the 16-bit address, the Decoder ensures that only the appropriate chip is activated for any given memory access, allowing the system to manage and access the 64K address space efficiently.

**Figure VI.4:** 3-to-8 Decoder using two 2-to-4 Decoders.




Adapted from: Bhagat:85.

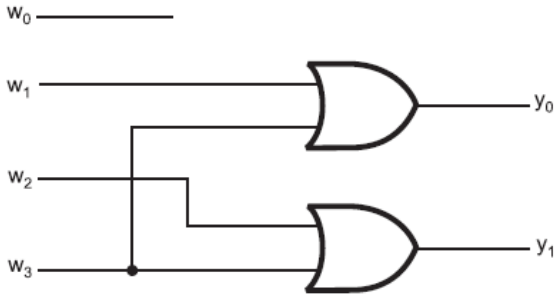
There are several different types of decoders. Binary to Decimal decoders convert binary input codes into their equivalent decimal representations and are commonly used in applications where binary data needs to be translated into a human-readable format (Mano & Kime, 2014; Arroz, Monteiro, & Oliveira, 2019). Binary to Binary decoders, for example, simply decode binary input combinations into specific binary output lines and are often used in address decoding circuits, as shown in the previous example, or in control logic to interpret control signals and activate specific functions or operations based on the input combination.

In a complementary way, an encoder performs the opposite function, i.e., it converts one or more input signals into a single encoded output.

A binary Encoder converts information from $2^n$ inputs into an *n-bit* code. Only one of the input signals should be 1 at any time, and the outputs represent the binary number corresponding to the active input. The truth table for a 4-to-2 encoder is shown in Figure VI.3 below. In this table, the output $y_0 = 1$ when either input $w_1$ or $w_3$ is 1, and output $y_1 = 1$ when input $w_2$ or $w_3$ is 1.

**Figure VI.7:** Logic circuit and truth table of a 4-to-2 Encoder.

| $w_3$ | $w_2$ | $w_1$ | $w_0$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |




Source: Bhagat:83.

It is assumed that the inputs are *one-hot encoded*, meaning only one input is 1 at any time. Input combinations with multiple 1s are not included in the truth table and are treated as *don't-care conditions*. Encoders are used to minimize the number of bits required to represent information. They are practical for transmitting data in digital systems because fewer wires are needed. Encoding is also beneficial for storage, as it reduces the number of bits that need to be stored.

There are different types of encoders. Priority encoders are designed to prioritize input signals, with the highest-priority signal taking precedence in the output and are commonly used in applications where input signals need to be ranked based on their importance. Decimal to Binary Encoders, on the other hand, convert decimal numbers into binary-coded outputs and are mostly used in applications such as digital display systems and data transmission.

# APPENDIX VII: GRADIENT DESCENT

## VII. Gradient Descent

A simple example with only two data points is given, using gradient descent to find the best fit line for these points. To keep things manageable, just a few iterations will be performed and small numbers will be used.

### Example and Step-by-Step Solution:

Given two data points: $(x_1, y_1) = (1,2)$, $(x_2, y_2) = (2,4)$ and $(x_3, y_3) = (3,4)$. A line $y = \phi_0 + \phi_1 x$ will be fitted using gradient descent. The parameters at the beginning are: $\phi_1 = 0, \phi_0 = 0$, and the chosen learning rate is $\alpha = 0.1$.

Then, the cost function will be defined. For this example, the cost function $L(\phi_1, \phi_0)$ for Mean Squared Error (MSE) will be used, given by (VII.1), where $n$ is the number of data points.

$$L(\phi_1, \phi_0) = \frac{1}{n}\sum_{i=1}^{n}\left((\phi_1 x_i + \phi_0) - y_i\right)^2 \quad \text{(VII. 1)}$$

For each iteration, the partial derivatives regarding $\phi_1$ (VII.2) and $\phi_0$ (VII.3) need to be computed.

$$\frac{\partial L}{\partial \phi_1} = \frac{2}{n}\sum_{i=1}^{n}\left((\phi_1 x_i + \phi_0) - y_i\right)x_i \quad \text{(VII. 2)}$$

$$\frac{\partial L}{\partial \phi_0} = \frac{2}{n}\sum_{i=1}^{n}\left((\phi_1 x_i + \phi_0) - y_i\right) \quad \text{(VII. 3)}$$

Then, for each iteration, the predictions and gradients will have to be computed; and $\phi_1$ and $\phi_0$ will have to be updated using gradient descent.

Starting with the first iteration and dividing it into three stages:

- **First Iteration**:

i. Compute Predictions

For $x_1 = 1: \widehat{y_1} = \phi_1 x_1 + \phi_0 = 0 * 1 + 0 = 0$

For $x_2 = 2: \widehat{y_2} = \phi_1 x_2 + \phi_0 = 0 * 2 + 0 = 0$

For $x_3 = 3: \widehat{y_3} = \phi_1 x_3 + \phi_0 = 0 * 3 + 0 = 0$

We will replace directly the parcel $(\phi_1 x_i + \phi_0)$ for its result on both calculations.

ii. Compute Gradients

$$\frac{\partial L}{\partial \phi_1} = \frac{2 * \big((0-2) * 1 + (0-4) * 2 + (0-4) * 3\big)}{3} = -\frac{44}{3}$$

$$\frac{\partial L}{\partial \phi_0} = \frac{2 * \big((0-2) + (0-4) + (0-4)\big)}{3} = -\frac{20}{3}$$

iii. Update Parameters

$$\boldsymbol{\phi_1}: \phi_1 - \alpha \frac{\partial L}{\partial \phi_1} = 0 - 0.1 * \left(-\frac{44}{3}\right) = \frac{22}{15} = 1.46(6)$$

$$\boldsymbol{\phi_0}: \phi_0 - \alpha \frac{\partial L}{\partial \phi_0} = 0 - 0.1 * \left(-\frac{20}{3}\right) = \frac{2}{3} = 0.6(6)$$

At the end of the first iteration, we have $\boldsymbol{\phi_1} = 1.46(6)$ and $\boldsymbol{\phi_0} = 0.6(6)$.

- **Second Iteration**

i. Compute Predictions

For $x_1 = 1$: $\widehat{y_1} = \phi_1 x_1 + \phi_0 = \frac{22}{15} * 1 + \frac{2}{3} = \frac{32}{15}$

For $x_2 = 2$: $\widehat{y_2} = \phi_1 x_2 + \phi_0 = \frac{22}{15} * 2 + \frac{2}{3} = \frac{18}{5}$

For $x_3 = 3$: $\widehat{y_3} = \phi_1 x_3 + \phi_0 = \frac{22}{15} * 3 + \frac{2}{3} = \frac{76}{15}$

We will replace directly the parcel $(\phi_1 x_i + \phi_0)$ for its result on both calculations.

ii. Compute Gradients

$$\frac{\partial L}{\partial \phi_1} = \frac{2 * \left(\left(\frac{32}{15} - 2\right) * 1 + \left(\frac{18}{5} - 4\right) * 2 + \left(\frac{76}{15} - 4\right) * 3\right)}{3} = \frac{76}{45}$$

$$\frac{\partial L}{\partial \phi_0} = \frac{2 * \left(\left(\frac{32}{15} - 2\right) + \left(\frac{18}{5} - 4\right) + \left(\frac{76}{15} - 4\right)\right)}{3} = \frac{8}{15}$$

iii. Update Parameters

$$\phi_1 : \phi_1 - \alpha \frac{\partial L}{\partial \phi_1} = \frac{22}{15} - 0.1 * \left(\frac{76}{45}\right) = \frac{292}{225} = 1.297(7)$$

$$\phi_0 : \phi_0 - \alpha \frac{\partial L}{\partial \phi_0} = \frac{2}{3} - 0.1 * \left(\frac{8}{15}\right) = \frac{46}{75} = 0.613(3)$$

At the end of the second iteration, we have $\phi_1 = 1.297(7)$ and $\phi_0 = 0.613(3)$.

Further iterations will be computed in the same way, gradually refining $\phi_1$ and $\phi_0$ to minimize the MSE, the cost function, bringing the line closer to the best fit for our data points. By iterating this process several more times, values of $\phi_1$ and $\phi_0$ that yield the lowest possible error could be eventually reached.

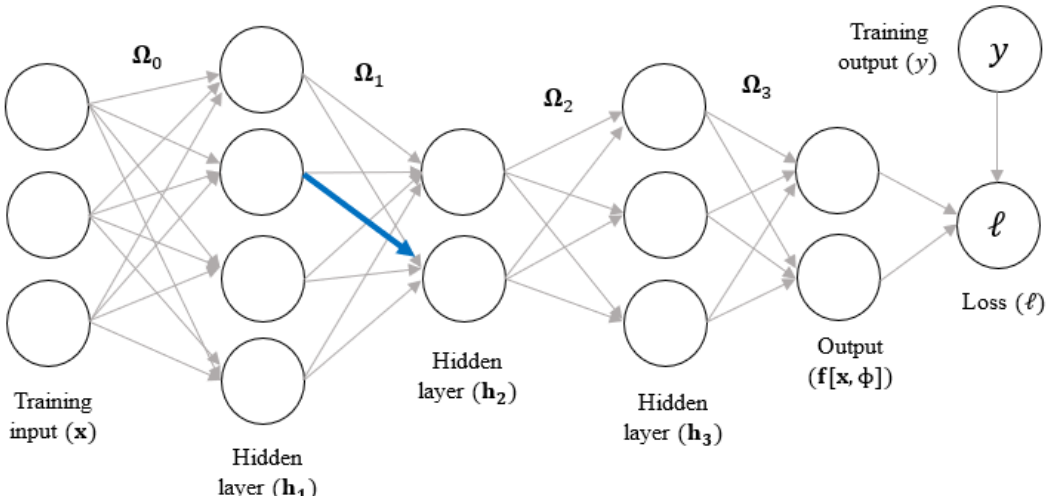
# APPENDIX VIII: BACKPROPAGATION ALGORITHM

## VIII. Backpropagation Algorithm

Stochastic gradient descent is a powerful optimization technique used to update the parameters of ML models, including NNs. However, as the complexity of NNs grows, with multiple layers and numerous interconnected neurons, efficiently calculating the gradients becomes crucial. This is where Backpropagation comes in.

While stochastic gradient descent focuses on how to update parameters by using a subset of data points to compute gradients, backpropagation provides the mechanism to compute those gradients efficiently, particularly in deep networks. By applying the chain rule to calculate the derivatives backwards through the network, backpropagation ensures that each layer's parameters can be adjusted in a computationally efficient manner, making it essential for training deep NNs.

**Figure VIII.1:** Backpropagation forward pass.




Adapted from: Prince, 2023:98.

Our goal with this algorithm is to calculate the derivatives of the loss function $\ell$, with respect to each weight (the arrows in Figure VIII.1 above) and bias. Essentially, the goal is to understand how small adjustments to each parameter will influence the loss.

Each weight connects one hidden unit to another, scaling the activation of the unit it's coming from and affecting the unit it's connected to. Thus, a small change in the weight will

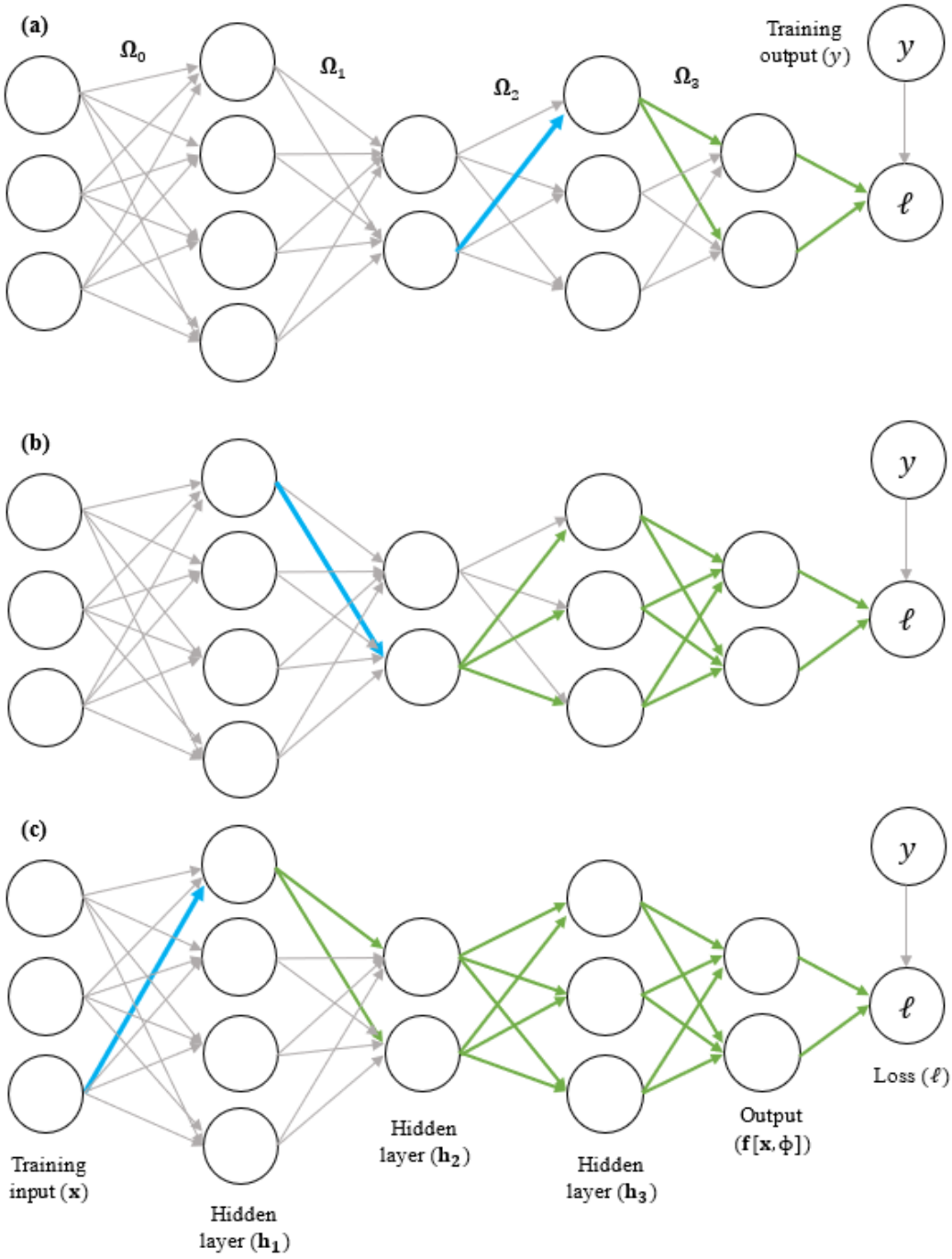
have an impact proportional to the activation of the source unit. For instance, if the activation of a hidden unit doubles, the influence of a small adjustment to the corresponding weight will also double. To calculate the derivatives of the weights, the activations need to be tracked at each hidden layer. This process is called the *forward pass* because it involves moving through the network layer by layer.

When adjusting weights in a NN, it is important to understand how these changes affect the final loss. As illustrated in Figure VIII.2 below, here's how it works for different layers:

- **(a)** layer $\boldsymbol{h}_3$: To see how changing a weight in $\boldsymbol{h}_3$ (indicated by the blue arrow) impacts the loss, it is important to trace how adjustments in this layer affect the final output $f$, and then see how $f$ impacts the loss (shown by the green arrows).
- **(b)** layer $\boldsymbol{h}_2$: For a weight in $\boldsymbol{h}_2$ (blue arrow), it is pivotal to first check how changes in this layer affect $\boldsymbol{h}_3$, then how $\boldsymbol{h}_3$ affects $f$, and finally how $f$ influences the loss (all indicated by the green arrows).
- **(c)** layer $\boldsymbol{h}_1$: Similarly, to understand the effect of a weight in $\boldsymbol{h}_1$ (blue arrow), it is crucial to look at how $\boldsymbol{h}_1$ influences $\boldsymbol{h}_2$, then follow how changes propagate through to affect the final loss (represented by the green arrows).

This process, called the *backward pass*, starts by calculating derivatives at the output and then works backwards through the network, taking advantage of the fact that many of these computations are reused.

**Figure VIII.4:** Backpropagation backward pass.




Adapted from: Prince, 2023:99.

Considering a network $\boldsymbol{f}[\boldsymbol{x}, \boldsymbol{\phi}]$, where $\boldsymbol{x}$ is a multivariate input, $\boldsymbol{\phi}$ the network parameters, and $\boldsymbol{h}_1, \boldsymbol{h}_2$ and $\boldsymbol{h}_3$ the three hidden layers of the network and the function a[•] applies the activation function individually to every component of the input (VIII.1).

$$\begin{aligned}
\boldsymbol{h}_1 &= \mathrm{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \boldsymbol{x}] \\
\boldsymbol{h}_2 &= \mathrm{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \boldsymbol{h}_1] \\
\boldsymbol{h}_3 &= \mathrm{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \boldsymbol{h}_2] \\
\boldsymbol{f}[\boldsymbol{x}, \phi] &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \boldsymbol{h}_3
\end{aligned} \tag{VIII. 1}$$

The parameters of the model, $\phi = \{\boldsymbol{\beta}_0, \boldsymbol{\Omega}_0, \boldsymbol{\beta}_1, \boldsymbol{\Omega}_1, \boldsymbol{\beta}_2, \boldsymbol{\Omega}_2, \boldsymbol{\beta}_3, \boldsymbol{\Omega}_3\}$ include the bias vectors $\boldsymbol{\beta}_k$ and weight matrices $\boldsymbol{\Omega}_k$ for each of the $K$ layers, in this case, three layers.

Additionally, as previously seen, there is also an individual loss terms $\ell_i$, based on the model's prediction $f[\boldsymbol{x}_i, \boldsymbol{\phi}]$, for the training input $\boldsymbol{x}_i$. The total loss is calculated by summing these individual terms across the entire training dataset (VIII.2).

$$L[\phi] = \sum_{i=1}^{I} \ell_i \tag{VIII. 2}$$

Then, the stochastic gradient descent is applied as the optimization algorithm for training this NN, updating the parameters as (VIII.3), where $\alpha$ is the learning rate and $\boldsymbol{\mathcal{B}}_t$ incorporates the batch indices at the $t^{th}$ iteration.

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \boldsymbol{\mathcal{B}}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \tag{VIII. 3}$$

However, in order to compute (VIII.3), the gradients $\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k}$ and $\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k}$ have to be calculated for the parameters $\{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$ at every layer $k \in \{0, 1, \ldots, K\}$, as well as for every index $i$ in the batch. For this the backpropagation algorithm is applied, consisting of three steps: the forward pass (Figure VIII.1 above) and two Backward passes (Figure VIII.2 above). For the purposes of this example, the ReLU function is the activation function, a[•].

The backpropagation algorithm starts with the forward pass, by describing the network as a set of sequential calculations (VIII.4), where $\boldsymbol{f}_{k-1}$ is the pre-activations at the $k^{th}$ hidden layer (the values before the application of the activation function) and $\boldsymbol{h}_k$ is a vector containing the activations at the $k^{th}$ hidden layer (the values after applying this activation function). The term $\mathrm{l}[\boldsymbol{f}_3, y_i]$ represents the loss function that can be, for example, the least squares.

$$\boldsymbol{f}_0 = \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \boldsymbol{x}_i$$

$$\boldsymbol{h_1} = \mathrm{a}[\boldsymbol{f_0}]$$

$$\boldsymbol{f}_1 = \boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \boldsymbol{h}_1$$

$$\boldsymbol{h_2} = \mathrm{a}[\boldsymbol{f_1}]$$

$$\boldsymbol{f}_2 = \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \boldsymbol{h}_2$$

$$\boldsymbol{h_3} = \mathrm{a}[\boldsymbol{f_2}]$$

$$\boldsymbol{f}_3 = \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \boldsymbol{h}_3$$

$$\ell_i = \mathrm{l}[\boldsymbol{f}_3, y_i] \qquad \text{(VIII. 4)}$$

In the forward pass of backpropagation, each of the intermediate variables $f_k$ and $h_k$ are computed and saved until our loss is finally calculated.

Nevertheless, when the pre-activations $\boldsymbol{f}_0, \boldsymbol{f}_1, \boldsymbol{f}_2$ are modified, the loss function changes depending on how these pre-activations influence the final output of the network. Thus, applying the chain rule, (VIII.5) is obtained as the expression for the gradient of the loss, $\ell_i$, with respect to $\boldsymbol{f}_2$.

$$\frac{\partial \ell_i}{\partial \boldsymbol{f}_2} = \frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{f}_2} \frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{h}_3} \frac{\partial \ell_i}{\partial \boldsymbol{f}_3} \qquad \text{(VIII. 5)}$$

In the same way, the change in loss is calculated regarding the changes in $\boldsymbol{f}_1$ and $\boldsymbol{f}_0$ (VIII.6).

$$\frac{\partial \ell_i}{\partial \boldsymbol{f}_1} = \frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{f}_1}\frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{h}_2}\left(\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{f}_2}\ \frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{h}_3}\ \frac{\partial \ell_i}{\partial \boldsymbol{f}_3}\right)$$

$$\frac{\partial \ell_i}{\partial \boldsymbol{f}_0} = \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{f}_0}\frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{h}_1}\left(\frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{f}_1}\ \frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{h}_2}\ \frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{f}_2}\ \frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{h}_3}\ \frac{\partial \ell_i}{\partial \boldsymbol{f}_3}\right) \quad \text{(VIII. 6)}$$

Note that each term in the calculations is based on results from previous steps. By moving *backward* through the network, these previously computed values can be reused. This method is efficient because it avoids redundant calculations.

Taking (VIII.5) and starting by the term $\frac{\partial \ell_i}{\partial \boldsymbol{f}_3}$ which is the derivative of the loss $\ell_i$ with respect to the network's output $\boldsymbol{f}_3$. This derivative typically has a straightforward form that depends on how the loss function is defined. Then, in the same equation, there is the term $\frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{h}_3}$, the derivative of the network output $\boldsymbol{f}_3$ with respect to the hidden layer activations $\boldsymbol{h}_3$, that is given by (VIII.7), where, $\boldsymbol{\Omega}_3$ is the weight matrix for this layer and the result is the transpose of this weight matrix.

$$\frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{h}_3} = \frac{\partial}{\partial \boldsymbol{h}_3}(\boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\boldsymbol{h}_3) = \boldsymbol{\Omega}_3^{\mathrm{T}} \quad \text{(VIII. 7)}$$

Finally, still in (VIII.5) above, there is the term $\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{f}_2}$, the derivative of the activation $\boldsymbol{h}_3$ with respect to its input $\boldsymbol{f}_2$ and is determined by the activation function. For activation functions like ReLU, this derivative is a diagonal matrix where each diagonal entry is either 0 (for negative inputs) or 1 (for non-negative inputs). To simplify computations, instead of working with this diagonal matrix directly, a vector $\mathbb{I}[\boldsymbol{f}_2 > 0]$ is used to indicate where the input values are positive and perform pointwise multiplication.

This same logic is applied to (VIII.6). Moving backward through the network, it is pivotal to alternate between multiplying by the transpose of the weight matrices, $\boldsymbol{\Omega}_k^{\mathrm{T}}$ (which are stored during the forward pass) and applying thresholds based on the input values, $\boldsymbol{f}_{k-1}$, to the hidden layers. This approach ensures that each step of the *backward pass* efficiently uses the information from the forward pass and simplifies the calculation of gradients.

This is the first *backward pass*. In essence, the backpropagation algorithm leverages previously computed values and uses efficient matrix operations to update weights and biases throughout the network.

After this first backward pass, computing the derivative of the loss, $\frac{\partial \ell_i}{\partial \boldsymbol{f_k}}$, is clear. It is now important to turn the attention to finding the derivatives of the loss with respect to the weights and biases. To find the derivative of the loss with respect to the biases $\boldsymbol{\beta}_k$, the chain rule (VIII.8) is used, that was in fact already calculated in (VIII.5-VIII.6).

$$\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} = \frac{\partial \boldsymbol{f}_k}{\partial \boldsymbol{\beta}_k}\frac{\partial \ell_i}{\partial \boldsymbol{f}_k} = \frac{\partial}{\partial \boldsymbol{\beta}_k}(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \boldsymbol{h}_k)\frac{\partial \ell_i}{\partial \boldsymbol{f}_k} = \frac{\partial \ell_i}{\partial \boldsymbol{f}_k} \tag{VIII. 8}$$

In the same way, for the weight's matrix, $\boldsymbol{\Omega}_k$, the derivative is (VIII.9).

$$\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} = \frac{\partial \boldsymbol{f}_k}{\partial \boldsymbol{\Omega}_k}\frac{\partial \ell_i}{\partial \boldsymbol{f}_k} = \frac{\partial}{\partial \boldsymbol{\Omega}_k}(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \boldsymbol{h}_k)\frac{\partial \ell_i}{\partial \boldsymbol{f}_k} = \frac{\partial \ell_i}{\partial \boldsymbol{f}_k}\boldsymbol{h}_k^T \tag{VIII. 9}$$

In this case, the result is a matrix that has the same dimensions as $\boldsymbol{\Omega}_k$ and depends linearly on $\boldsymbol{h}_k$, that was multiplied by $\boldsymbol{\Omega}_k$ in the original equation (these values were computed during the forward pass). This matches the initial idea that the derivative of the weights in $\boldsymbol{\Omega}_k$ should relate directly to the values of the hidden units $\boldsymbol{h}_k$ they influence. In other words, how much a weight's adjustment affects the loss is proportional to the activation level of the hidden units it interacts with. And so, the second final *backward pass* is completed.

# APPENDIX IX: PROBABILISTIC T-NORM AND T-CONORM OPERATIONS

## IX. Probabilistic T-norm and T-conorm Operations

In probabilistic and fuzzy logic systems, the concept of truth is more flexible than in traditional binary logic. While binary logic only uses two values (0, 1), probabilistic and fuzzy logic allow truth values to range between [0; 1], representing varying degrees of truth or likelihood. For example, a statement might be 70% true, or have a 0.7 probability of being true, rather than being strictly true (1) or false (0).

To handle these values, it is important to generalize logical operations that extend the basic operations of AND and OR to work with probabilities. These operations are essential when working with uncertain or imprecise information.

T-norms, or triangular norms, are used in probabilistic logic to represent the AND operation, i.e., as the mathematical equivalent of the logical AND in real-valued logic. They calculate what's called the *expected conjunction*, which is the probability that all inputs are true at the same time. Essentially, T-norms help us combine multiple probabilities to determine how likely it is that multiple events will occur together.

A T-norm is a binary operation $\top: [0,1] * [0,1] \rightarrow [0,1]$, that must satisfy four key properties:

- *Associativity*, meaning that the grouping of values doesn't affect the result:$\top\big(x, \top(y, w)\big) = \top(\top(x, y)w)$.
- *Commutativity*, meaning the order of the operands doesn't change the outcome:$\top(x, y) = \top(y, x)$.
- *Monotonicity*, where the result follows the same order if one operand is less than or equal to another:$x \leq w) \wedge (y \leq q) \Rightarrow \top(x, y) \leq \top(w, q)$.
- *The neutral element property*, where combining a value with 1 leaves the result unchanged:$\top(x, 1) = x$

These operators are applied in probabilistic AND operations to combine two or more probabilities (Petersen *et al.*, 2023).

These properties ensure that T-norms behave in a way that aligns with basic logic at the edges of the [0,1] range. For T-norms, which represent the logical AND, the properties imply that (1,1) maps to 1, while pairs like (0,1) or (1,0) map to 0. By the same logic, (0,0) also maps to 0 due to monotonicity, meaning that combining smaller values results in a smaller outcome (Petersen *et al.*, 2023).

On the other hand, T-conorms are used in probabilistic logic to represent the OR operation, calculating what's called the *expected disjunction*. This gives us the probability that at least one of the inputs is true. T-conorms help determine how likely it is that at least one event from a set of possibilities will happen.

T-conorm serves as the mathematical equivalent of the logical OR in real-valued logic. Like T-norms, T-conorms operate on two values between 0 and 1 and produce a result within the same range. T-conorms are binary operations $\perp: [0,1] * [0,1] \rightarrow [0,1]$, that must also satisfy specific properties:

- *Associativity*, which means that grouping the operands in any way doesn't change the result:$\perp \big(x, \top \perp (y, w)\big) = \perp (\perp (x, y)w)$.
- *Commutativity*, meaning the order of the operands doesn't affect the outcome:$\perp (x, y) = \perp (y, x)$.
- *Monotonicity*, where if one operand is greater than or equal to another, the result will follow the same order:$(x \leq w) \wedge (y \leq q) \Rightarrow \perp (x, y) \leq \perp (w, q)$.
- *The neutral element property*, which leaves the result unchanged when a value is combined with 0: $\perp (x, 0) = x$

These properties ensure T-conorms behave consistently within a real-valued logic system, making them essential for probabilistic OR operations in models where probabilistic outcomes are combined (Petersen *et al.*, 2023).

These generalized operations play a crucial role in differentiable NNs that incorporate probabilistic logic. When a model uses probabilistic logic, it can handle smooth gradients during training. Gradients are important because they guide the model in adjusting its parameters to improve its performance. Unlike traditional binary activation functions (which

only output 0 or 1), probabilistic activation functions can output values between 0 and 1, allowing the model to represent uncertainty and compute expectations over probabilistic inputs. This makes training more effective, as it enables the model to handle uncertain, real-world data more naturally (Petersen *et al.*, 2023).

# APPENDIX X: SCRIPT IN PYTHON

## X. Script in Python

The Python script used for the practical part of this dissertation can be found under:
https://github.com/FilipaTrindade/BooleanAlgebra__DeepLearning