UNIVERSITY OF THE AEGEAN

End-to-End Semantic Communication over Quantum-Cryptographic Secure Channel

Author:
Nikolaos BERMPARIS

Supervisor:
Dr. Konstantinos MALIATSOS

This Diploma thesis was submitted in fulfillment of the requirements of Bachelor of Science in the

Department of Information and Communication Systems Engineering

November 7, 2025

Declaration of Authorship

I, Nikolaos BERMPARIS, declare that this thesis titled, "End-to-End Semantic Communication over Quantum-Cryptographic Secure Channel" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

UNIVERSITY OF THE AEGEAN

Abstract

End-to-End Semantic Communication over Quantum-Cryptographic Secure Channel

by Nikolaos BERMPARIS

Semantic communication has become an increasing advent area in information transmission. With recent advances in deep learning such as Long Short-Term Memory (LSTM) networks, this thesis develops an end-to-end secure semantic communication framework that encodes natural language into compact semantic vectors. These vectors are modulated using Binary Phase Shift Keying (BPSK) and transmitted through a simulated Additive White Gaussian Noise (AWGN) channel implemented in MATLAB. The received signals are demodulated and decoded via an LSTM-based semantic decoder in Python.

Beyond the baseline communication pipeline this thesis also explores the vulnerabilities of semantic communication systems under adversarial conditions. In particular, we investigate the effect of gradient-based adversarial attacks such as the Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD). Carefully crafted perturbations can disrupt the semantic integrity of transmitted messages. With that in mind, experiments demonstrate that even small perturbations can induce significant semantic distortion, highlighting the critical need for robustness in practical deployments. The results provide insights into the susceptibility of LSTM-based autoencoders to adversarial manipulation, bridging the gap between classical communication noise models and modern adversarial threat landscapes.

In order to defend against these attacks the thesis integrates cryptographic mechanisms into the communication pipeline. More specifically a hybrid security layer which combines Quantum Key Distribution (QKD) with the use of the BB84 protocol and AES-based encryption. These measures ensure confidentiality and forward secrecy across the transmission of information while adversarial training and semantic correction methods strengthen resilience against gradient-based attacks. The findings have put to the test the security of semantic communication and what can be achieved by unifying advances in deep learning, cryptography and adversarial defense. Further we pave the way for future information systems where meaning is transmitted both reliably and securely across noisy and adversarial environments.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Konstantinos Maliatsos, for believing in me and giving me the opportunity to undertake this research. His guidance, support, and trust throughout this journey have been invaluable, and this work would not have been possible without his encouragement.

I would also like to sincerely thank my parents, whose unwavering support and belief in me since day one have been the foundation of all my academic and personal achievements. Their love and sacrifices continue to inspire me every day.

Contents

List of Figures

List of Tables

Chapter 1

Intoduction

1.1 Fundamental Knowledge

In recent decades there have been rapid technological advancements which reshape the way we communicate. At the same time, advancements in the field of Artificial Intelligence have transformed our lives in ways that we couldn't even imagine. Modern communication systems like 6G networks are already integrating AI into their architectures while deep model integrations are offering numerous solutions in a large variety of fields.

However, from a security standpoint, things remain constantly critical as malicious actors figure out ways to exploit and disrupt these systems. It has always been a continuous struggle between the new technological systems and security. In particular, with the implementation of neural networks in systems, a new form of attacks has evolved, also known as "adversarial attacks," since they have posed one of the greatest concerns. It has been proven that with input injections they can manipulate the neural networks into providing wrong outputs, backdoor access, and more. The concept of semantic communication (SemCom) offers a promising alternative to traditional communication paradigms. Rooted in the foundational work of Shannon and Weaver in classical information theory, semantic communication was first envisioned by Weaver in the 1940s although he could not fully pursue it due to his collaboration with Shannon's focus on bit-level transmission. Conventional bit-based communication (BitCom) aims at transmitting data with maximum fidelity at the bit level which ensures that received bits match with the transmitted ones. On the other hand, semantic communication shifts the focus from bits to meaning—emphasizing the accurate delivery of intended information, even if the bit-level representation is altered. Recurrent model architectures such as long short-term memory (LSTM) networks have the ability to capture context, dependencies, and semantics within data streams without encountering the vanishing gradient problem (as seen in other RNN architectures). As a result, SemCom can provide a more efficient, low-spectrum communication proving that the intended meaning rather than the exact bit sequence is preserved. Notwithstanding its advantageous attributes, SemCom failed to attract adequate attention for an extended duration. The reasons may be twofold. Shannon stated that "the semantic dimensions of communication are inconsequential to the engineering facets" since the significance of communications can be associated with particular physical and conceptual entities. Semantic transmission hence invariably impacts the mathematical model's generality Weaver (1949). Furthermore, because of the much greater urgency of high-rate, reliable communications at that time, researchers primarily adopted the Shannon-established classical information theory (CIT) Shannon (1948). Recent work has noted that, in the absence of semantic awareness, simply ensuring a symbolic recurrence can be considered a way to preserve syntactic information regardless. The semantic information theory (SIT), also

known as the classical semantic information theory (CSIT), was proposed by Carnap et al. as an analogue of the CIT.

These types of systems help in terms of effectiveness of frameworks as they enhance performance across different fields such as industrial IoTs, UAV systems, autonomous vehicles, and smart healthcare. What they do is they focus on task-relevant semantics which works more efficiently in the long run.

Modern networks are in a paradigm shift coming from traditional syntactic communication (focused purely on transmitting bits accurately) to semantic communication, which prioritizes the transmission of meaningful information. This shift is driven by the emerging 6G applications as well as deep learning advancements. From autonomous driving to intelligent healthcare, machines must exchange information not just to reconstruct raw messages but to enable correct inferences or actions in real-time. Unlike conventional designs rooted in Shannon's classic information theory, which we will cover in the next section, semantic communication operates at the semantic level, concentrating on what the data means to the receiver. This means that essential semantics are extracted from a message and only what is relevant to the recipient's task or context is sent. A semantic-aware system can significantly reduce the amount of data transmitted while still preserving the message's intent. In fact, the receiver side can often interpret or act on the information even if some bits are corrupted, since the core meaning is retained. The result is a communication paradigm that substantially lowers bandwidth usage and enhances reliability, meeting the needs of future intelligent networks. This transformation is so fundamental that it is seen as a key enabler for 6G systems. In conclusion, semantic communication marks a move from "transmit then understand" to now a smarter "understand-then-transmit" approach—just as more efficient and context-aware connectivity is possible than ever before with purely syntactic transmission. In the next section, we will dive into real-life applications of SemCom.

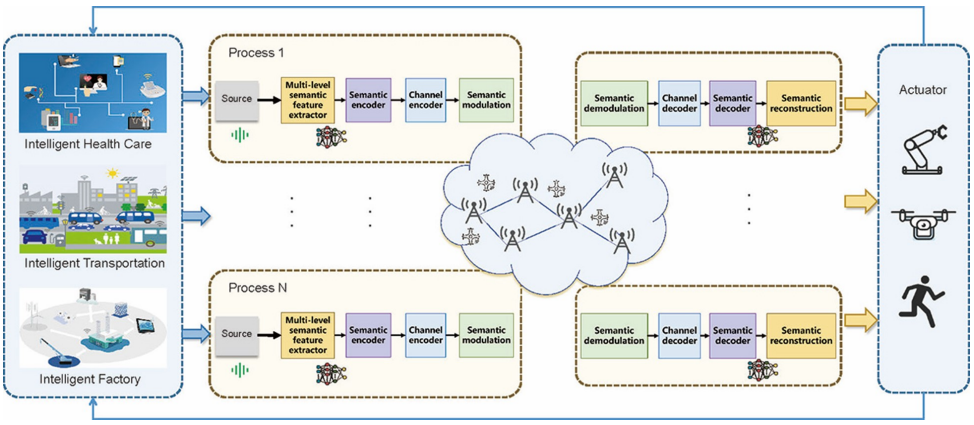

FIGURE 1.1: Semantics-empowered goal-oriented networks.

1.2 Industrial IoTs (IIoTs)

In industrial IoTs, semantic frameworks improve different factors such as efficiency, reliability, and accuracy. Such systems are used in intelligent transportation, smart grids, and industrial automation. On the other hand, in this sector, new problems emerge such as energy efficiency and the value of the transmitted data. For instance, by using RNN LSTM models, the transmission overhead is reduced (for sensors,

for example). By using semantic-aware data processing and model trading, IIoT systems reach more efficient and improved communication leading to better performance.

In Industrial IoT environments such as smart factories and infrastructure monitoring, semantic communication promises to address the massive data deluge and real-time control requirements far better than traditional methods. Future IIoT scenarios (especially under 6G networks) will involve huge numbers of intelligent devices and sensors that demand ultra-reliable, low-latency links (sub-millisecond delays, minimal jitter, and near-zero packet loss) for time-critical operations. Syntactic communication of raw sensor readings in such settings would overwhelm networks; instead, semantic communication can dramatically streamline data flows. Industrial IoT devices and edge nodes have the ability to pre-process and transmit only high-level "meanings" or anomalies relevant to the task at hand. For example, a network of industrial sensors might share an alert like "pressure drop in valve X exceeds threshold" instead of continuous pressure readings. This approach aligns with the SemComIIoT initiative, where edge AI algorithms are employed at the network edge to extract relevant features and filter out unessential information, injecting only the meaningful data into the network. By conveying data about data (i.e., metadata and insights) rather than raw streams, semantic communication in IIoT not only reduces bandwidth consumption but also enables faster, goal-driven responses (such as automated shutdowns or maintenance requests) in complex industrial systems. Early research prototypes confirm these benefits; semantic encoding schemes have been shown to cut down data transmissions while still supporting deterministic control requirements in Industry 4.0 settings. This makes semantic communication an attractive paradigm for IIoT, where efficient, intelligent exchanges can power next-generation automation and industrial intelligence.

1.3 Unmanned Aerial Vehicles (UAVs)

Semantic communication is also proving valuable in networks of Unmanned Aerial Vehicles - for instance, in disaster response, surveillance, and environmental monitoring by drone swarms. UAVs often capture high-volume sensor data (like video or images) and operate under strict constraints (limited onboard energy, variable wireless links). Semantic techniques can enable these systems to transmit mission-critical insights without sending raw data. A notable example is a wildfire detection system using UAVs with deep learning-based semantic communication. In this system, the drone's camera feed is processed through a deep joint source-channel coding (DJSCC) model (essentially an AI-driven semantic codec) that encodes only the essential features of images relevant to fire detection. Instead of streaming full-resolution video, the UAV sends compact semantic representations (for example, the presence and location of smoke or flame). At the ground station, a corresponding decoder or classifier can directly use these semantic features to decide if there is a wildfire, often without needing to reconstruct the original image. This semantic approach yielded tangible benefits: it achieved lower latency, higher reliability, and improved energy efficiency compared to traditional image transmission methods. In practice, DJSCC-based semantic transmission allowed the UAVs to operate longer and send alerts faster, as less data was sent over noisy links. Beyond wildfire monitoring, similar semantics-empowered UAV communication prototypes have been proposed for tasks like real-time mapping and target recognition. All these efforts demonstrate that by communicating what the UAV observes (the meaning) rather

than raw sensor feeds, one can significantly enhance the effectiveness of UAV networks in time-sensitive and bandwidth-limited operations.

1.4 Smart Cities and IoT Sensor Networks

Smart cities are starting to invest in a large number of IoT sensors, cameras, and connected devices to manage urban services such as traffic control, public safety, and more. However, these devices can overwhelm communication networks and back-end processing due to their data requirements during transmission. Semantic communication offers a powerful solution by ensuring that only contextually relevant information is exchanged across the city's networks. Specifically, in a realistic scenario, this could mean that city infrastructure would communicate "semantics" of the sensed data (condensed descriptions of events or conditions) instead of continuous raw data streams. For example, a number of environmental sensors might collaboratively report "air quality normal in all regions" or pinpoint "anomalous pollution spike at a specific location" instead of each sensor having to send raw particulate measurements every minute. Additionally, a network of surveillance cameras could share alerts like "suspicious motion detected in a parking lot" without streaming all video feeds. Such context-driven data exchange dramatically reduces redundant traffic, allowing city operators to focus on meaningful insights. Academic studies have proven that semantic communication can maintain the integrity and usefulness of information across diverse urban applications. In IoT smart city scenarios, where a vast variety of sensors and actuators often need to exchange contextual information, semantic techniques (by encoding that contextual meaning) can provide an understanding based on each subsystem—from smart homes to traffic lights—while data interpreters are placed in the correct context in order to enable interoperability and proactive responses.

1.5 Satellite and Space Communications

Satellite and space-based communications (the "Internet of Space") present extreme networking challenges. These include long delays, intermittent connectivity, limited spectrum, and energy-constrained spacecraft, where conventional bit-focused communication often falters. Here too, semantic communication is emerging as a transformative paradigm, enabling more efficient and mission-aware information exchange in non-terrestrial networks. The idea is to transmit "mission-relevant" semantics (context-aware summaries of observations or status) instead of raw telemetry or imagery, thereby making the most of each precious contact opportunity between satellites or between a rover and Earth. Recent work proposes semantic architectures tailored for space missions, showing that by intelligently encoding and sending only the most relevant information, one can overcome many space communication limitations. For example, a Mars rover might semantically encode a dust storm observation as a small set of parameters based on severity, location, and trend rather than send unprocessed high-resolution images of the storm. This way, the rover can conserve bandwidth and energy while still transmit crucial knowledge to mission control. In fact, a deep-space semantic communication experiment involving Mars dust storm monitoring demonstrated extraordinary gains: the semantic system was able to achieve up to 50× more data transmissions per unit of energy (per battery life) compared to a traditional raw-data system. In that scenario, conventional methods would have exhausted the spacecraft's energy long before

the 180-day mission requirement, whereas the semantic approach comfortably met the duration by drastically reducing redundant bits. This example demonstrates how semantic communication can directly tackle severe bandwidth and power constraints in space. With that in mind, we can send fewer, more meaningful bits as satellites transmit more updates and make better use of contact windows. Beyond Mars rovers, researchers envision semantic communication in space applications from satellite swarms that share summarized situational awareness to deep-space probes that adaptively report only scientifically significant findings. With that focus on context and intent, semantic techniques promise to greatly enhance the efficiency and autonomy of future space communication networks, which is why they are being actively explored as part of 6G non-terrestrial network (NTN) innovations.

(I) Information theory fundamentals

The origins of *SemCom* can be seen in Weaver's groundbreaking work from the 1950s Weaver (1949), when he postulated three categories of communication problems:

- How precisely can communication symbols be conveyed?
- To what extent do the communicated symbols accurately express the intended meaning?
- To what extent does the meaning that has been received influence behaviour in the desired manner?

Thankfully, Shannon's classical information theory Shannon (1948), which has been used as a reliable guide for communication system design for more than 70 years, is thought to adequately address the first technical issue.

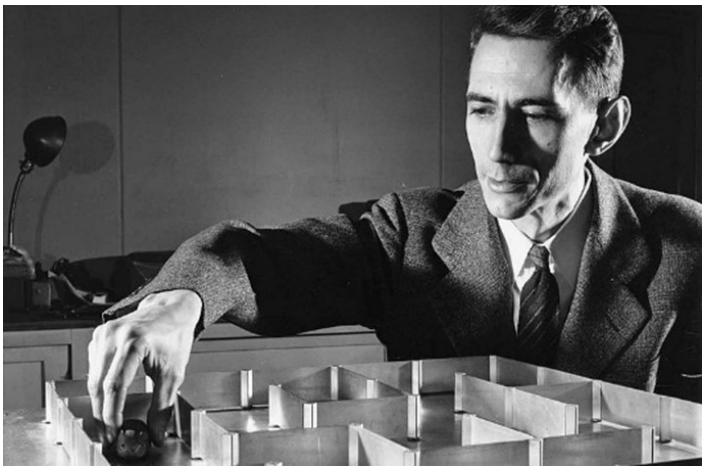

FIGURE 1.2: Image of Claude Elwood Shannon.

Shannon's groundbreaking contributions to information theory addressed two essential inquiries within digital communication theory: what is the maximum attainable data compression (the entropy H), and what constitutes the maximum feasible communication transmission rate (the channel capacity C). Shannon proposed the idea of entropy as a metric for information. By employing probability theory, one can measure the information content of a message or signal in bits. Information

entropy serves a dual purpose: it quantifies the amount of information and defines the theoretical limits for data compression and transmission rates. The average code length for practical encoding cannot be lower than the theoretical minimum established by information entropy.

For discrete variables, let $X \in \{x_1, x_2, \ldots, x_n\}$ be a source characterized by the probability mass function $p(x_i)$, where $x_i \in X$. The entropy $H(X)$ of the random variable X is defined in the following way:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i)$$

The channel capacity is the upper limit of mutual information that can be conveyed via the channel. One can view mutual information as the decrease in uncertainty about one random variable based on knowledge of the other. The channel capacity, with respect to the input X and the output Y, is defined by:

$$C = \max_{p(x)} I(X;Y)$$

where $I(X;Y)$ is the mutual information between X and Y.

The lossy source coding theorem, known as the rate-distortion theory, can be used to determine the minimal number of bits per symbol, measured by the rate R, for a given distortion D^*. Similarly, the rate-distortion function $R(D^*)$ represents the lower bound of the transmission rate for a given maximum average distortion D^*, which is defined by

$$R(D^*) = \min_{D \leq D^*} I(X;Y)$$

Where:

$$D = \sum_{x,y} p(x)\, p(y|x)\, d(x,y)$$

is the *expected distortion* between input x and output y.

- $I(X;Y)$ = mutual information between X (source) and Y (reconstruction),
- $d(x,y)$ = distortion metric, with $d(x,y) = 0$ when $x = y$,
- D^* = maximum allowable distortion,
- $R(D^*)$ = minimum rate (bits per symbol) required to achieve average distortion $\leq D^*$.

Chapter 2

Theoretical Background

2.1 Fundamental Knowledge

2.1.1 Introduction to Semantic Communication

Classical Semantic Information Theory

Classical Semantic Information Theory (CSIT) originates from the foundational work of Claude Shannon, who introduced the mathematical framework of information theory by incorporating the concept of meaning into communication. His work established the groundwork for modern digital communication systems.

The notion of **channel capacity** quantifies how much information a channel can reliably transmit, relating directly to the semantic content of messages and their relevance to the receiver. In 1948, Shannon published his seminal paper *"A Mathematical Theory of Communication"*, which provided the first mathematical definition of information and demonstrated that data could be transmitted accurately through noisy communication channels such as wireless links or telephone lines. These revolutionary concepts paved the way for the modern information age.

What Is Semantic Communication (SemCom)?

The basic philosophy behind it comes from the assumption that common knowledge is shared. This is something that is supposed to happen before the basic communication takes place. Let's suppose that we want to transmit a message (as mentioned in the paper, message x) it first has to be encoded. Then after the transmission phase, the receiver decodes the message and then compares the inference to the knowledge base. A message x can represent different types of media from text to audio and other methods as well. As an example we can compare this system to the enigma as it used to scramble a message to protect it's meaning, then transmit it through radio (our physical transmission equivalent) and then the receiver deciphers the message.

As seen in the current section, an explanation for SemCom could be based on each block where the whole process involves the sequence below:

The idea behind it is so that all bits are accurately sent but we need to make sure that these are the "meaningful" bits or the so-called "semantic information". This important mention consists of the Warren Weaver model of communication which is a 3 level model that consists of the technical level, semantic level and effectiveness level accordingly. It is an intelligent method since the data with the true meaning are being transported.

From our understanding what we should also keep in mind is the semantic similarity. As we saw in the beginning when we transmit information, at the end of the day we care how much the received message has been altered to the original transmitted. In this case what we aim to achieve is the unaltered meaning between the

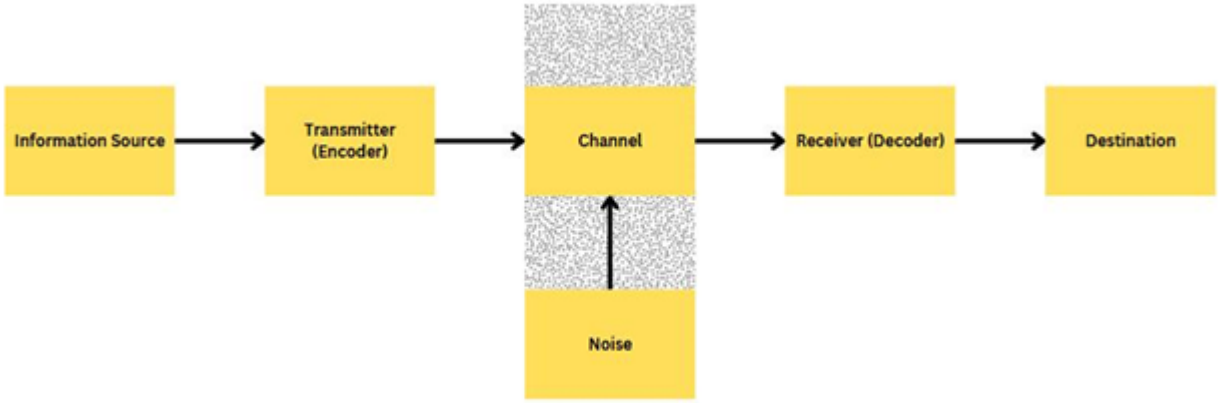

FIGURE 2.1: The essential parts of the Shannon-Weaver model.

transmission and reception of the message. After that we measure how many bits we saved (a term mentioned as compression ratio).

In the EL-SemCom we focus on how well the transmitted meaning helps in order to achieve a specific task or a goal.

Components of SemCom

Some of the components and structure of a typical semantic communication start from high-level conceptual design to practical. The authors Xie et al. (2022) reference Weaver's communication model and emphasize that SemCom primarily focuses on how and what to transmit in a semantically efficient manner. In our case, the goal isn't to transmit all bits with maximum accuracy but rather to convey the essential semantic meaning. This allows for significant reductions in bandwidth by omitting semantically redundant content.

A critical distinction of SemCom is its reliance based on the corresponding knowledge base (KB) which provides both for the sender and receiver to interpret messages consistently.

Semantic Representations

Most mentioned semantic information theories are simple and limited to low dimensional scenarios. Classical semantic similarity measurements are based on certain rules such as tree or graph based semantic representations as they are not feasible for real world applications.

Classical approaches to semantic similarity were mentioned like classical semantic similarity measurements which rely on structured rules (e.g. graph-structured semantic representations). Besides that, statistical metrics such as term frequency-inverse document frequency (TF-IDF) and information content work as a count for similarity. Also more advanced techniques were mentioned like n-gram contexts, syntactic and lexical customization, and machine learning-based methods.

The main problem is that while structured representations work well for simple semantic concepts, it is hard to generalize across complex relationships and varied meanings. With that in mind deep learning approaches can be used to handle these difficulties with flexible similarity measurements.

2.2 Semantic Channel Capacity

The difference in **SemCom** however is the way it *focuses on the semantic similarity between the input and output messages*. For a comparison of the transmitted and received messages the metric below is presented as:

$$\frac{H(S \mid X)}{H_s(Y)} \tag{2.1}$$

Where 2.1 represents the ratio of $H(S \mid X)$ conditional entropy to some entropy-like quantity $H_s(Y)$.

From this point we do not look over the traditional syntax accuracy but towards a preservation of the information meaning. This is not correspondent towards only physical channels but also semantic noise. It is important to keep a consistent meaning for our information as their functionality relies on bit discrepancies shared in knowledge bases.

The concept of Semantic Channel Capacity works as an extension of Shannon's theory while we are headed towards a semantic fidelity.

2.2.1 Knowledge Base

SemCom introduces the concept of a Knowledge Base (KB), which is considered a compact information entity that houses the background knowledge relevant to a specific application domain (like text or sound) associated with a particular type of SemCom service. This idea stems from the fact that, due to the vastness of knowledge Xie et al. (2022); Xia (2024), it is unthinkable to represent all knowledge within a single framework. The structure of a KB can, in a general sense, encompass various computational ontologies, as well as facts, rules, and constraints related to a particular domain. In SemCom systems driven by Deep Learning (DL), the background knowledge consists of training data samples that serve a specific class of learning tasks. Given the requirements for computation and storage, Base Stations (BSs) and user equipment with large capacity in the wireless cellular network can maintain certain amounts and kinds of KBs. For these users with low capacity, a possible option is to obtain various SemCom services with the necessary KBs by linking up with different BSs. Utilizing the knowledge sharing method to gain the necessary background information from neighboring Mobile Users (MUs) or Base Stations is another feasible tactic, though it will result in additional preparation delays.

2.2.2 Joint Source-Channel Coding in SemCom

Formal setup & notation

Let a text source produce a length-m token sequence $S^m = (S_1, \ldots, S_m)$ with alphabet $\mathcal{S}$ and distribution p_S. We transmit it over a memoryless channel $p_{Y|X}$ using n channel uses Shannon (1948); Cover and Thomas (2006).

- **Source-channel rate:** $r \frac{m}{n}$ (tokens per channel use).
- **Code:** encoder $f_{m,n} : \mathcal{S}^m \to \mathcal{X}^n$ and decoder $g_{m,n} : \mathcal{Y}^n \to \hat{\mathcal{S}}^m$.
- **Distortion (sequence level):** Classical additive distortion measure Cover and Thomas (2006); Berger (1971):

$$d(S^m, \hat{S}^m) = \frac{1}{m}\sum_{i=1}^{m} d(S_i, \hat{S}_i) \quad \text{(classical additive case)} \tag{2.2}$$

or a semantic sequence metric $d_{\text{sem}}(S^m, \hat{S}^m)$ (e.g., embedding loss).
A pair (r, D) is achievable if there exists a sequence of (m, n) codes with

$$\limsup_{m\to\infty} \mathbb{E}\left[d\left(S^m, g_{m,n}(Y^n)\right)\right] \le D. \tag{2.3}$$

2.2.3 Separation baseline: rate-distortion and capacity

The classical rate–distortion (RD) function Shannon (1959); Berger (1971); Cover and Thomas (2006) is defined as

- **Rate-distortion (RD):**

$$R(D) = \inf_{p(\hat{S}|S):\mathbb{E}[d(S,\hat{S})]\le D} I(S;\hat{S}) \tag{2.4}$$

The capacity of a discrete memoryless channel (DMC) is given by Shannon (1948); Cover and Thomas (2006):

- **Channel capacity:**

$$C = \sup_{p_X} I(X;Y) \tag{2.3}$$

Separation theorem (asymptotic, memoryless setting, additive distortion):

$$rR(D) < C \iff (r, D) \text{ achievable.}$$

This is the modular "compress-then-protect" benchmark. It's not generally optimal at finite blocklengths or for non-additive/semantic distortion.

Distortion models for text & useful RD formulas

(A) Token-level accuracy (Hamming)

For a binary memoryless source with Hamming distortion, the rate–distortion function is (Cover and Thomas, 2006, Example 13.3.2):

$$R(D) = 1 - H_b(D), \quad 0 \le D \le \tfrac{1}{2}, \quad H_b(u) = -u\log_2 u - (1-u)\log_2(1-u).$$

(B) Embedding/latent MSE (semantic proxy)

Mapping each token to a continuous Gaussian latent vector $Z \in \mathbb{R}^k$ with $Z_j \sim \mathcal{N}(0, \sigma_j^2)$ gives an additive MSE distortion model. The Gaussian rate–distortion theorem Berger (1971); Cover and Thomas (2006); Gallager (1968) yields

$$R_j(D_j) = \frac{1}{2}\log_2\left(\frac{\sigma_j^2}{D_j}\right), \quad 0 < D_j \le \sigma_j^2,$$

and by water-filling,

$$R(D) = \sum_j R_j(D_j), \qquad D = \sum_j D_j.$$

(C) Cosine-based semantic distortion

With unit-norm embeddings $\varphi(\cdot)$, cosine similarity is defined as Manning and Schütze (1999)

$$d_{\cos}(S, \hat{S}) = 1 - \langle \varphi(S), \varphi(\hat{S}) \rangle.$$

For small angular deviations between unit vectors, a second-order approximation Manning and Schütze (1999) gives

$$d_{\cos}(S, \hat{S}) \approx \frac{1}{2} \| \varphi(S) - \varphi(\hat{S}) \|^2,$$

which locally reduces the cosine loss to an MSE form, connecting semantic embedding distortion to the Gaussian RD model.

Channel Models

The AWGN channel or Additive White Gaussian Noise is a widely used model in communication simulation and information theory. It is a basic noise model used in information theory that mimics the effect of many random processes that occur in nature. This noise arises from random processes and exhibits a constant power spectral density, making it "white" across all frequencies. AWGN channels are often used to simulate real-world scenarios, where noise can corrupt the signal during transmission Xie (2021).

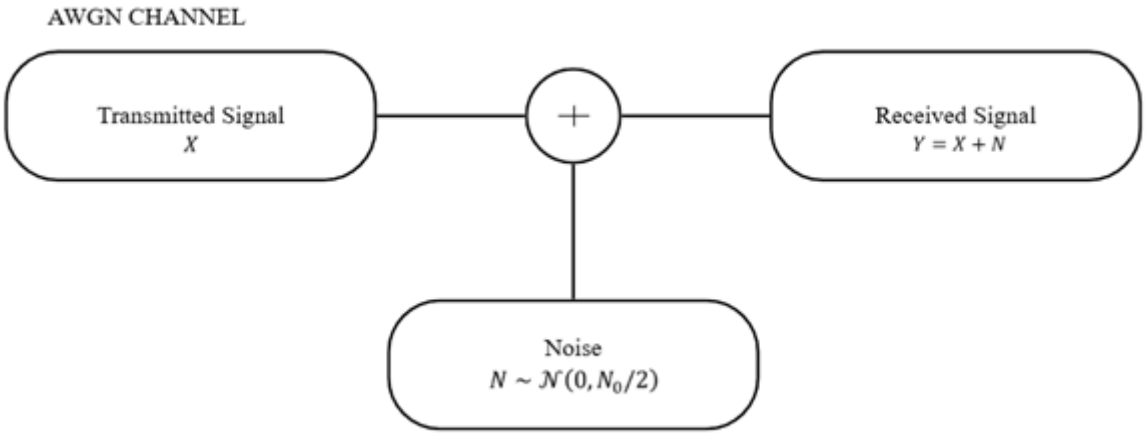

FIGURE 2.2: AWGN channel figure.

Binary Phase Shift Keying or BPSK is a modulation method that is utilized in digital wireless communications. It is a modulation technique with two levels. It is the binary signals 0s and 1s that are indicated by two distinct phase shifts, typically 180 degrees apart. BPSK is applicable in different forms of digital communication systems like 802.11, GSM and CDMA. It is a form of modulation in which every bit is depicted by a two-level waveform. BPSK is one of the simplest types of digital modulation because it features only two levels and does not require any coding. The sent bits are readily decoded which makes it ideal for secure and dependable data transmission. The main benefit of BPSK is its simplicity. The transmitter only has to alternate between the two phase states whereas the receiver is solely tasked with

measuring the phase at its end. This allows for a straight forward implementation and carries a minimal risk of error. Generally BPSK represents a dependable and straightforward modulation technique that is suitable for our implementation.

(A) BPSK over AWGN (per real channel use)

Transmit $X \in \{-\sqrt{P}, +\sqrt{P}\}$, receive

$$Y = X + N, \qquad N \sim \mathcal{N}(0, N_0/2).$$

- **Bit error probability (hard decision):**

$$P_b = Q\left(\sqrt{\frac{2P}{N_0}}\right) = Q(\sqrt{2\mathrm{SNR}}), \tag{2.5}$$

where $Q(\cdot)$ is the standard *Gaussian Q-function*, i.e., the tail probability of a standard normal variable:

$$Q(x) = \Pr(Z > x), \qquad Z \sim \mathcal{N}(0,1).$$

- **Capacity (real AWGN):**

$$C = \frac{1}{2}\log_2(1 + \mathrm{SNR}) \quad \text{bits/use}. \tag{2.6}$$

(For complex baseband AWGN, the factor $1/2$ is removed.)

(B) Effective BSC approximation

With BPSK + hard decision, you can approximate the channel as BSC(ε) with $\varepsilon \approx P_b$ from (2.8); then

$$C \approx 1 - H_b(\varepsilon).$$

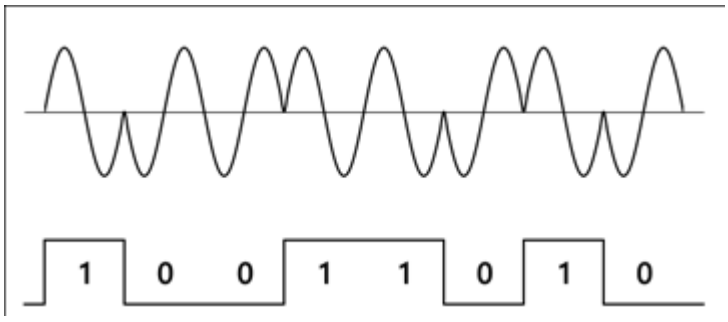

FIGURE 2.3: BPSK Modulation.

2.2.4 "Remote"/semantic JSCC (meaning over form)

Let $S^\star$ denote the semantic variable (meaning), tokens T are observations via $p_{T|S^\star}$. The decoder estimates $\hat{S}^\star$. For a semantic distortion $d(S^\star, \hat{S}^\star)$,

$$R_{\text{remote}}(D) = \inf_{p(S^\star|T):\mathbb{E}[d(S^\star,\hat{S}^\star)]\le D} I(T;\hat{S}^\star) \tag{2.7}$$

Then the separation test becomes $rR_{\text{remote}}(D) < C$. This formalizes: "send only what is needed for meaning."

2.2.5 Deep JSCC as constrained optimization (training view)

LSTM/Transformer + differentiable channel realizes a joint map

$$S^m \xrightarrow{f_\theta} X^n \xrightarrow{\text{channel}} Y^n \xrightarrow{g_\phi} \hat{S}^m.$$

We want (i) semantic fidelity, (ii) fit a bandwidth/power budget, (iii) be robust across SNRs:

$$\min_{\theta,\phi} \mathbb{E}\left[\underbrace{\mathcal{L}_{\text{sem}}(S^m, \hat{S}^m)}_{\text{e.g., -cosine, CE, BLEU surrogate}}\right] + \lambda \underbrace{\mathbb{E}\|X^n\|^2}_{\text{power}} + \mu \underbrace{\mathcal{L}_{\text{robust}}}_{\substack{\text{SNR sampling,} \\ \text{adv. noise}}}$$

$$\text{s.t.} \quad \frac{1}{n}\mathbb{E}\|X^n\|^2 \le P_0, \quad \frac{m}{n} = r.$$

A **Lagrangian** view ties (2.11) to the RD-capacity tradeoff: $\mathcal{L}_{\text{sem}}$ pushes toward small D (thus large $R(D)$), while the power/bandwidth terms limit effective C.

2.2.6 Finite-blocklength guidance (why JSCC helps in practice)

At practical m, n, the normal approximations give (schematically)

$$\text{Source coding:} \quad mR(D) \approx mR - \sqrt{mV_S}Q^{-1}(\epsilon_S)$$

$$\text{Channel coding:} \quad nC \approx nC - \sqrt{nV_C}Q^{-1}(\epsilon_C)$$

where V_S, V_C are dispersions, $\epsilon_{S,C}$ small error/overflow probabilities. A separated system must satisfy both bounds independently, which is looser than a truly joint mapping-hence JSCC's empirical gains at modest blocklengths.

2.2.7 Classical JSCC baselines (useful anchors)

(A) Binary at $r = 1$ over BSC

With Hamming distortion and $X = S$ (uncoded), $\hat{S} = Y$:

$$D = \Pr[\hat{S} \neq S] = \varepsilon.$$

Separation also gives $D = \varepsilon$ at $r = 1 \Rightarrow$ uncoded is optimal in this corner case.

(B) Analog Gaussian match at $r = 1$

Scalar $Z \sim \mathcal{N}(0, \sigma^2)$, real AWGN with SNR. A linear analog map achieves

$$D = \frac{\sigma^2}{1 + \text{SNR}}, \quad R(D) = \frac{1}{2}\log_2\left(\frac{\sigma^2}{D}\right) = \frac{1}{2}\log_2(1 + \text{SNR}) = C.$$

Perfect Shannon-Kolmogorov "matching"; a strong intuition for latent-JSCC with MSE.

2.2.8 Mapping to your pipeline (LSTM, BPSK, metrics)

- Your encoder learns $Z = f_\theta(S^m)$ (latent). If training uses embedding/MSE/-cosine, you're approximating the Gaussian RD regime in (2.6)/(2.14).
- BPSK/AWGN links to (2.8)–(2.9). With hard decisions you can sanity-check using BSC with $\varepsilon \approx Q(\sqrt{2\mathrm{SNR}})$.
- Report BLEU / WER (sequence-level) and embedding cosine (semantic) to quantify D under various SNRs; these are practical surrogates for d_{sem}.

An expert system in the realm of artificial intelligence (AI) is a computer system that mimics the decision-making capabilities of a human expert. Expert systems are built to tackle intricate issues by reasoning through knowledge bases primarily represented as if–then rules, rather than using traditional procedural programming code. Among the earliest genuinely successful types of AI software were expert systems. They originated in the 1970s and spread rapidly in the 1980s, after which they were widely seen as the future of AI.

2.2.9 DeepNN / E2E / Deep JSCC / DeepSC

A Deep Neural Network (DNN) is a machine learning model that simulates the way the human brain processes information. DNNs can learn patterns from data and make predictions based on past experiences, much like humans, in contrast to traditional algorithms that operate according to predefined rules. Deep Neural Networks (DeepNN) and End-to-End (E2E) learning have helped semantic communication by enabling joint optimization of encoding and decoding processes. Deep Joint Source-Channel Coding (Deep JSCC) and Deep Semantic Communication (DeepSC) use deep learning to extract, compress, and transmit semantic information directly, bypassing traditional modular designs. These are ideal for complex settings.

From first sight semantics is traditionally related to natural language (text based). Semantic communication of text can offer interesting insights into the benefits of DeepJSCC. In particular, the transformer architecture, which has proven highly effective for natural language processing (NLP) tasks, can be used to extract semantic features beneficial for JSCC. Its potential to outperform separate source compression followed by channel coding has been recognized for a long time. Nonetheless, designing practical joint coding schemes has proven to be a considerable challenge because of the intricacies of optimizing source and channel coding at the same time.The emergence of DL has brought a shift in JSCC, which enables the development of more efficient and effective joint coding schemes. With various methodologies we can adapt to various channel conditions without explicit channel modeling. One of them is an autoencoder. It is a type of neural network that attempts to learn a compressed representation of its input data.

2.2.10 LSTM-Based Autoencoders

CNN (Convolutional Neural Network) A Convolutional Neural Network (CNN) is a neural network architecture used in Deep Learning and is commonly applied to Computer Vision tasks. Many different types of data, such as text, images, and audio, have been processed and predicted from using this kind of deep learning network. Due to the shared-weight architecture of the convolution kernels or filters that slide along input features and yield translation-equivariant responses called feature

maps, CNNs are also referred to as shift invariant or space invariant artificial neural networks.

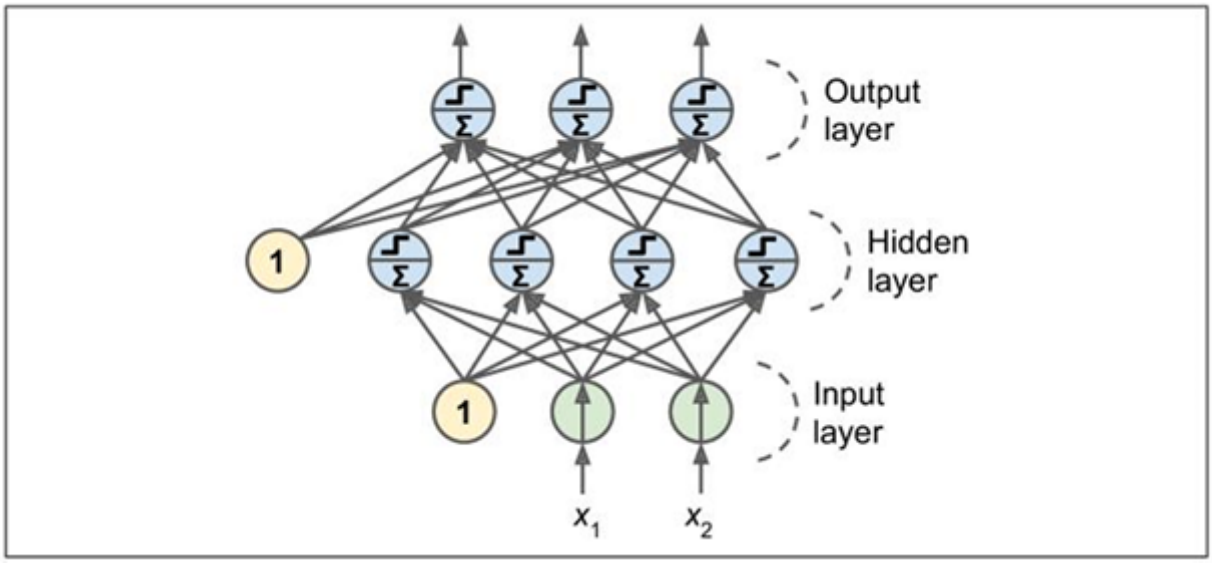

FIGURE 2.4: The Multi-Layer Perceptron.

With the development of backpropagation (BP) and multi-layer perceptrons (MLPs) the limitations of single-layer networks worked out and enabled training of deeper architectures. This advancement brought back interest in neural networks and laid the groundwork for deep learning.

2.2.11 RNN Baseline & Gradient Pathology

Unlike feedforward neural networks the output of a neuron at one time step is fed back as input to the network at the next time step. This enables RNNs to capture temporal dependencies and patterns within sequences. That way as well there is a feedback where the network learns from its past processes.

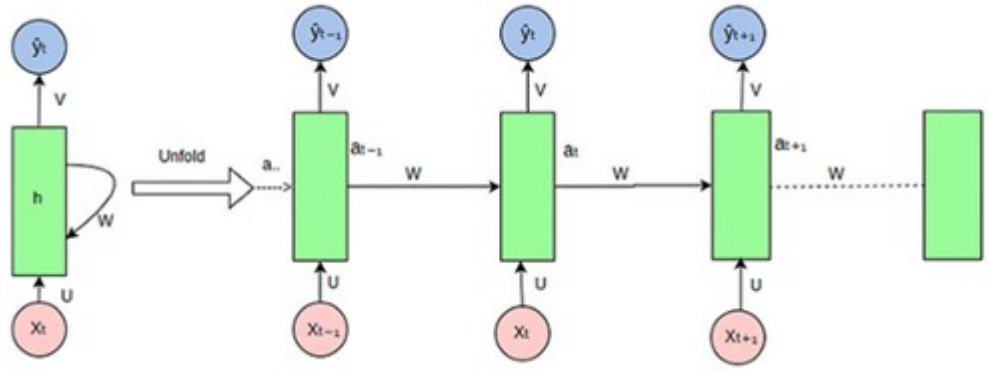

FIGURE 2.5: Unrolling Recurrent Neural Network.

However RNNs suffer from problems such as the vanishing gradient problem where the number of forward propagation steps in a network increases the gradients

of earlier weights are calculated with increasingly many multiplications. So after a while it might introduce instability in the training process, slow it, or halt it entirely. With that in mind the issue was addressed with the development of the long short term memory (LSTM) architecture.

GAN-based methods GAN is a method that has been used in many fields due to its usability such as realize real time adaptive compression. Some GAN based solutions can be extremely-effective in image compression while also the same system can fully synthesize unimportant regions in decoded messages. GANs or Generative Adversarial Networks excel at synthesizing photo-realistic images as they have a multi-purpose use in different fields such as motion transfer , image generation from drawings , VR etc Prijatelj et al. (2020).

2.2.12 LSTM Cell

Long short-term memory (LSTM) is a type of recurrent neural network (RNN) aimed at mitigating the vanishing gradient problem commonly encountered by traditional RNNs. It aims to provide a short-term memory for RNN that can last thousands of timesteps (thus "long short-term memory"). An LSTM unit is typically composed of a cell and three gates: an input gate, an output gate, and a forget gate. The cell remembers values over arbitrary time intervals, and the gates regulate the flow of information into and out of the cell. Forget gates decide what information to discard from the previous state, by mapping the previous state and the current input to a value between 0 and 1.

As mentioned by Mathworks "LSTM networks are a specialized form of the RNN architecture. RNNs use past information to improve the performance of a neural network on current and future inputs. They contain a hidden state and loops, which allow the network to store past information in the hidden state and operate on sequences. RNNs have two sets of weights: one for the hidden state vector and one for the inputs. During training, the network learns weights for both the inputs and the hidden state. When implemented, the output is based on the current input, as well as the hidden state, which is based on previous inputs" Peng et al. (2022, 2024).

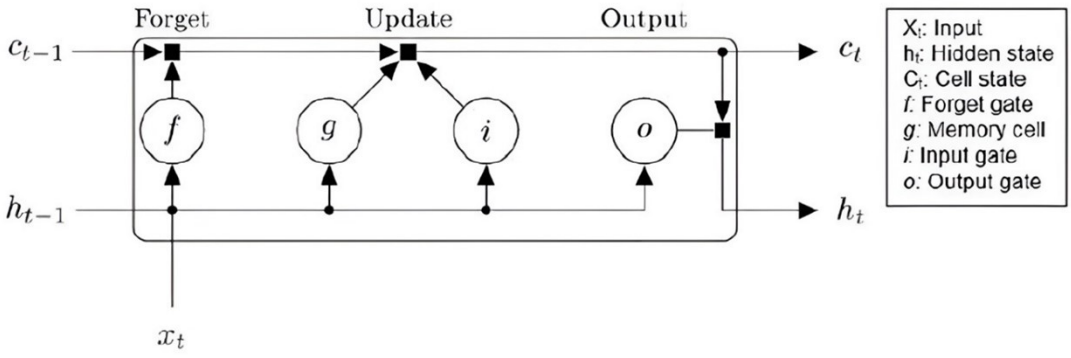

FIGURE 2.6: Long Short-Term Memory Cell.

What is an autoencoder?

At a fundamental level, autoencoders represent a kind of artificial neural network primarily utilized for unsupervised learning Their primary objective is to acquire a

compressed, or "encoded," representation of information and subsequently rebuild the original data from that compressed form. Consider it akin to mastering the art of condensing a lengthy article into a few concise sentences: the autoencoder aims to extract the essential information from the input and subsequently reconstructs it with high fidelity Prijatelj et al. (2020).

Semantic Encoder-Decoder (Autoencoder) Setup

Autoencoders generate a reconstruction of the original input. The autoencoder comprises two smaller networks: a decoder and an encoder. Throughout training the encoder acquires a collection of characteristics referred to as a latent representation from the input data. Simultaneously, the decoder is trained to rebuild the data using these characteristics. The autoencoder can subsequently be utilized to forecast inputs that have not been encountered before. Autoencoders are highly adaptable and can be applied to various types of data, such as images, time series, and text.

The architecture of an autoencoder consists of two main parts:

1. Encoder: The encoder processes the input data (such as an image, audio clip, or text file) and condenses it into a lower-dimensional representation referred to as the latent space or encoded representation.

2. Decoder: The decoder attempts to reconstruct the original input from the compressed representation. The objective is to reduce the disparity between the original data and the reconstructed data.

Autoencoder as referenced in Lu et al. (2024) Semantic communication conveys semantic messages that pertain to a series of correctly structured symbols acquired from the "meaning" of the source. The receiver thus aims to grasp completely the "meaning" of the encoded semantic symbols. Consequently a crucial element of semantic communications is to extract the source's meaning while disregarding superfluous information. Neural networks are usually used to extract semantics from the source because of their strong representation and learning capabilities. In particular, the autoencoder as a type of neural network has the ability to learn efficient representations for high-dimensional data by extracting its most important information. This makes it especially suitable for semantic communications. Specifically, the autoencoder comprises an encoder and a decoder. The encoder generates encoded symbols that have significantly fewer dimensions than the source data, as it retains only the key information and discards the insignificant parts of the data. The decoder is then employed to retrieve the original data from the low-dimensional symbols. Utilizing autoencoders for extracting semantic information offers the following benefits. To begin with, autoencoders can be built using different kinds of neural networks, including convolutional neural networks, transformers, and fully-connected neural networks. Consequently, it can be used for semantic communications involving various types of source data, such as text, images, videos, and multi-modal data. Furthermore, the transmission efficiency of semantic communications can be greatly enhanced due to the fact that the encoder's output has far fewer dimensions. Furthermore, autoencoders lend themselves to self-supervised training. Nevertheless, autoencoders also present some major challenges. Specifically, while machines can efficiently comprehend the coding produced by the autoencoder, it remains beyond human understanding. This situation appears to contradict the principle of semantic communications to some extent.

2.3 Semantic Similarity Metrics

BLEU (Bilingual Evaluation Understudy)

The BLEU Papineni et al. (2002) metric was originally proposed for automatic evaluation of machine translation. It measures the degree of overlap between candidate translations and one or more reference translations. BLEU relies on modified n-gram precision and introduces a brevity penalty to penalize overly short translations.

The BLEU score is defined as:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

Where:

- N: the highest order of n-gram considered (commonly $N = 4$)
- w_n: weight for each n-gram precision (usually $w_n = \frac{1}{N}$)
- p_n: modified *n-gram precision*
- BP: *brevity penalty* to penalize overly short sentences

BERTScore

BERTScore Zhang et al. (2019) uses contextual embeddings from transformer-based models (e.g., BERT, RoBERTa) to compute similarity between sentences. Instead of relying on surface-level n-grams, it compares embeddings of tokens.

The BERTScore similarity between sentences x and y is computed as:

$$\text{BERTScore}(x, y) = \frac{1}{|x|} \sum_{i=1}^{|x|} \max_j \cos(\mathbf{e}_i^x, \mathbf{e}_j^y),$$

where $\mathbf{e}_i^x$ is the embedding of token i in sentence x, and $\cos(\cdot, \cdot)$ denotes cosine similarity. Variants exist for precision, recall, and F1-score.

Word Error Rate (WER)

The Word Error Rate (WER) Wikipedia (2024b) is a standard metric in speech recognition and natural language processing. It measures transcription accuracy based on the edit distance (Levenshtein distance) between a reference and a hypothesis sequence.

Word error rate can then be computed as:

$$\text{WER} = \frac{S + D + I}{N},$$

Where:

- S is the number of substitutions,
- D is the number of deletions,
- I is the number of insertions,
- C is the number of correct words,
- N is the number of words in the reference ($N = S + D + C$).

Cosine Similarity

Cosine similarity Levenshtein (1966) is a measure of similarity between two non-zero vectors by computing the cosine of the angle between them. It is widely used for comparing semantic embeddings.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \, \|\mathbf{B}\|},$$

where $\mathbf{A}$ and $\mathbf{B}$ are vectors, $\cdot$ denotes the dot product, and $\| \cdot \|$ represents the Euclidean norm.

t-SNE (t-Distributed Stochastic Neighbor Embedding)

t-SNE van der Maaten and Hinton (2008) is a nonlinear dimensionality reduction technique often used for visualization of high-dimensional data. It seeks to preserve local neighborhood structure by minimizing the Kullback–Leibler divergence between probability distributions in high- and low-dimensional spaces.

$$\mathrm{KL}(P \parallel Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}},$$

where p_{ij} is the probability of similarity between points i and j in high-dimensional space, and q_{ij} is the corresponding probability in the low-dimensional embedding.

UMAP (Uniform Manifold Approximation and Projection)

UMAP McInnes et al. (2018) is a dimensionality reduction technique grounded in Riemannian geometry and algebraic topology. It constructs a fuzzy topological representation of the data manifold and then optimizes its low-dimensional embedding. The objective function minimizes the cross-entropy between high- and low-dimensional fuzzy simplicial sets:

$$C = \sum_{(i,j)} \left[w_{ij} \log \frac{w_{ij}}{w'_{ij}} + (1 - w_{ij}) \log \frac{1 - w_{ij}}{1 - w'_{ij}} \right],$$

where w_{ij} are the high-dimensional edge weights and w'_{ij} are the low-dimensional equivalents.

2.4 Security in Semantic Communication

Modern communication systems need to ensure security while maintain certain principles. These specific principles exist in the well known CIA triad as they need to ensure the confidentiality, integrity, and availability of transmitted information since it is a fundamental requirement. In a realistic scenario the channel may be exposed to interception, tampering, or noise. Traditional cryptographic approaches such as AES have the mathematical security against unauthorized access, the characteristics of the communication environment often dictate how these methods are applied and combined with other protection mechanisms.

Eavesdropping is not the only thing to look out for when creating a secure channel but also safeguard against data manipulation, replay attacks, and degradation of transmitted meaning. In the context of semantic communication as we have mentioned transmitted content represents meaning rather than exact bit by bit symbols.

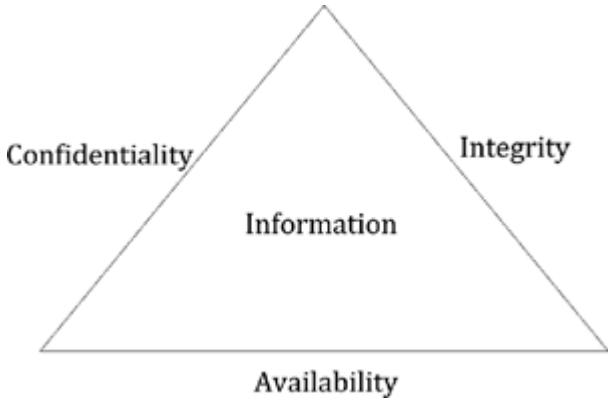

FIGURE 2.7: Classical CIA Triad.

What we need to keep safe is the semantic representation (latent vector) since it can be more efficient and potentially more resilient than securing the raw bitstream.

The AES Algorithm

General description of AES

The AES algorithm belongs to the category of Block Cipher algorithms. It was designed to replace the DES algorithm, which, due to its age (developed in the 1970s), was no longer considered secure. In AES, keys can be 128, 192, and 256 bits long. Depending on the length of the key used, we have the abbreviations AES-128, AES-192, and AES-256, respectively. Regardless of the key length, the algorithm always operates on 128-bit blocks. The encryption process is recursive, and each iteration is called a round. The number of rounds depends on the length of the key. In the first round, the algorithm uses a section of plaintext and the initial key as input. In subsequent rounds, the input is the block resulting from the previous encryption round and a key generated from the initial key based on a function defined by the algorithm. The output of the above process is an encrypted block (ciphertext). This block has exactly the same size (128 bits) as the plaintext block NIST (2001).

As already mentioned, AES takes as input sequences of 128 bits (blocks) and certain keys, which can be 128, 192, or 256 bits in size. These keys are called cipher keys in order to distinguish them from the keys generated during the operation of the algorithm. The processing of input data in AES is done at the byte level. Thus, the bits of a block or key are broken down into 8-bit segments to create bytes. Each byte in AES corresponds to a polynomial. If we symbolize the bits that make up a byte as b7, b6, b5, b4, b3, b2, b1, b0, then this byte is a representation of the polynomial:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0 = \sum_{i=0}^{7} b_ix^i$$

Encryption in the Advanced Encryption Standard (AES) takes place, as in any block cipher, through a number of rounds. The block of the original message — which serves as the input to the algorithm — is represented as a four-row table consisting of bytes. The four rows are numbered in order: 0, 1, 2, and 3. The number of columns (i.e., the width of the table) is determined by the size of the message segment.

In AES, where the message length is 128 bits, the number of columns in such a table is 4 (since $4 \times 4 \times 8 = 128$). In general, the number of columns in this state table is denoted by N_b. The same structure applies to the final ciphertext, as well as to the intermediate encryption process — all segments are represented as square tables of bytes. Each such table is referred to as a *state*.

Similarly, the input key is represented as a four-row table of bytes, where the number of columns N_k depends on the size of the key. For key sizes of 128, 192, and 256 bits, the respective values are $N_k = 4$, $N_k = 6$, and $N_k = 8$. The structure described below for both the data segment and the key is illustrated in Tables **??** and **??**, respectively, for the case of $N_b = N_k = 4$.

Status table

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

Key table

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Number of rounds in Rijndael

	$N_b = 4$	$N_b = 6$	$N_b = 8$
$N_k = 4$	10	12	14
$N_k = 6$	12	12	14
$N_k = 8$	14	14	14

The bytes of the status and key tables are filled in column by column. This means that the first 8 bits of the message occupy byte a0,0 of the initial state table, the next 8 bits occupy byte a1,0, and so on, while the key table is filled in accordingly. In this way, the first 8 bits of the encrypted segment occupy element a0,0 of the final state track, the next 8 bits occupy element a1,0, and so on Burmester et al. (2011).

The number Nr of rounds in Rijndael is found as a function of Nb and Nk.

In each round, four operations are performed in the following order:

- Byte substitution (SubBytes),
- Row shift (ShiftRows),
- Column mixing (MixColumns),
- AddRoundKey

where the output of each of these operations is the input for the next one. An exception is the last round, in which the MixColumns operation is not performed. Also, before the first round, the input segment is XORed with the key K. The entire encryption process is illustrated in the left part for the case Nr = 10. In general, AES does not have a Feistel structure, as in DES. It is also worth noting that the same procedures in each round (SubBytes, ShiftRows, MixColumns, AddRoundKey) are

also found in decryption. This simplifies the design of the algorithm, since only the procedures of each round, but with different substitution tables, are analyzed in decryption. In the rest of this section, the procedures are used in analogy with the formalism the hexadecimal system for representing bytes.

SubBytes process This stage constitutes the S-box of the algorithm: each byte of the state table is replaced by another byte from the table.

CBC (Cipher Block Chaining) Mode

In CBC mode, each plaintext block is combined using the XOR operation with the previous encrypted block. The result of the XOR operation is encrypted with the key. For the first XOR operation, an initial value called the Initialization Vector (Initialization Vector). The use of the XOR logical operation helps increase the security of the algorithm, as similar initial segments are covered. Although the process cannot be performed simultaneously, decryption can be done at the same rate as the code segment. Hence, every succeeding ciphertext block depends on the previous one. The initialization vector is of the same size as that of the plain text Burmester et al. (2011).

The Cipher Block Chaining (CBC) mode of operation which came in the year 1976 is the most commonly used and solves the security problem that exists in ECB. In this mode, each block of the message, before being encrypted, undergoes an XOR operation with the previous block of the ciphertext, i.e.

- **Encryption:**
$$C_i = E_K(M_i \oplus C_{i-1})$$
- **Decryption (for $i > 1$):**
$$M_i = D_K(C_i) \oplus C_{i-1}$$

Where:

- M_i : the plaintext block i
- C_i : the ciphertext block i
- $E_K(\cdot)$: encryption function under key K
- $D_K(\cdot)$: decryption function under key K
- $\oplus$: bitwise XOR

And the note says:
Therefore, for two identical plaintext blocks M_i, M_j, it holds that $C_i \neq C_j$. The CBC mode of operation is illustrated in Figure below Wikipedia (2024a).

To encrypt the first segment M_1M_1M1, where there is no encrypted segment to be added to it via an XOR operation, an initialization vector IV is used, whose length is equal to the length of the message segment, i.e.

$$C_1 = E_K(M_1 \oplus IV)$$

The same IV vector is also used in decryption, to recover. Both communicating parties must know the segment Wikipedia (2024a).

Although the secrecy is not mandatory, it is customary to protect it with ECB encryption. However, the integrity of the IV is of particular importance: if a third

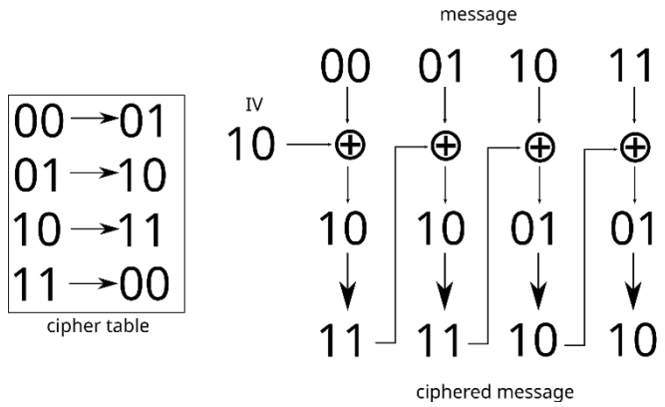

FIGURE 2.8: CBC encryption example with a toy 2-bit cipher.

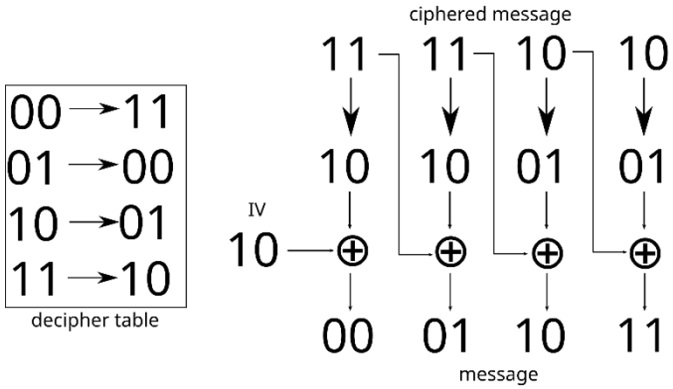

FIGURE 2.9: CBC example with a toy 2bits cipher.

party has the ability to modify it, then it can affect the first part of the message that the authorized recipient decrypts.

Furthermore, from the description of CBC, it is clear that there is a propagation of errors in two segments, i.e., if an encrypted segment is received with an error, then both it and the next one will be decrypted incorrectly (while in ECB only the current segment is affected). However, the subsequent encrypted sections will be decrypted correctly.

This is called the self-healing property of the CBC mode of operation.

(Essentially, this feature is a direct generalization of the one mentioned earlier, regarding the need to ensure integrity).

In other words, the attacker can introduce controlled changes to the message being decrypted.

This property is considered a weakness of the algorithm and has in some cases been used as a springboard for cryptanalysis. Particular attention should also be paid to the padding process, which can also be a source of cryptanalysis.

Quantum Key Distribution (QKD)

One of the most known applications of quantum cryptography is QKD. Quantum key distribution utilizes the unique properties of quantum mechanical systems to

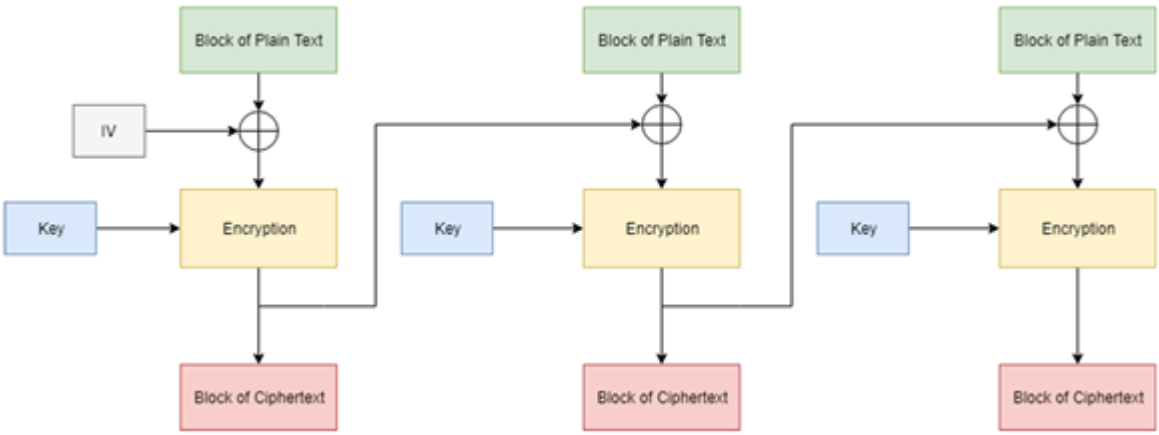

FIGURE 2.10: Encryption using CBC mode.

generate and distribute cryptographic keying material using special purpose technology.Specifically it allows two parties to generate a shared random secret key, which can then be used for encrypting and decrypting messages. The security of QKD is based on the principles of quantum mechanics, making it immune to computational attacks that can compromise traditional cryptographic keys.

In the context of SemCom QKD can be used to securely transmit semantic keys which ensures that only authorized parties can access the semantic information. A quantum SemCom framework that integrates QKD can enhance the security of semantic data transmission. So we will demonstrate that QKD can significantly reduce the risk of eavesdropping and ensure the confidentiality of semantic information.

To ensure this confidentiality and integrity in the semantic communication process a cryptographic layer was integrated into the transmission pipeline. This approach protects the semantically compressed information produced by the LSTM encoder before it is transmitted over a potentially insecure channel. The implementation utilizes the Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode to encrypt and decrypt the latent vectors. A shared symmetric key is simulated using Python's 'os.urandom()' function to emulate the secure, random key generation process characteristic of QKD systems Meng et al. (2025).

2.5 Adversarial and Data Poisoning Attacks

Background and Threat Model

Data poisoning attacks manipulate the training corpus to induce specific erroneous behaviors at inference while preserving high accuracy on clean inputs. In the targeted backdoor setting, the adversary injects training examples that bind a covert trigger pattern to an attacker-chosen output; the trained model behaves normally on standard inputs yet deterministically produces the attacker's target whenever the trigger appears. This paradigm, first systematized for deep vision models as "BadNets," established that tiny, stealthy perturbations during training can create persistent input-conditioned behaviors without degrading clean accuracy, motivating analogous studies in NLP.

NLP-Specific Backdoor Poisoning

Backdoor poisoning readily transfers to language tasks by using phrase-level triggers. Wallace et al. demonstrate *concealed data poisoning* for NLP models: adding a small number of crafted (input, label) pairs to the training data is enough to make the model predict an attacker-selected label whenever a chosen phrase occurs, while leaving standard validation performance essentially unchanged. Importantly, triggers can be natural-looking phrases embedded in otherwise benign sentences, improving stealth and making signature-based detection difficult

Formalization

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ be clean pairs and $\mathcal{P} = \{(x^\star \oplus \tau, y^\dagger)\}$ be the poisoned set with trigger τ. The training objective for model f_θ (LSTM encoder/decoder) is:

$$\min_{\theta} \frac{1}{N + |\mathcal{P}|} \sum_{(x,y)\in\mathcal{D}\cup\mathcal{P}} \mathcal{L}_{\mathrm{CE}}(f_\theta(x), y)$$

At test time, the Attack Success Rate (ASR) is $\Pr\{f_\theta(x^\star \oplus \tau) = y^\dagger\}$ over held-out carrier sentences $x^\star$ containing the trigger. High ASR with near-baseline clean accuracy is the hallmark of a successful targeted backdoor poisoning attack.

Backdoor Poisoning in Semantic Communication

Semantic communication systems rely on learned encoder–decoder architectures to transmit text as compressed latent vectors, often with additional channel coding such as our BPSK modulation and security mechanisms such as AES encryption and QKD-based key distribution. Unlike traditional symbol-based communication, the semantic layer directly maps input meaning into latent space representations. This paradigm, while efficient, is susceptible to *data poisoning attacks* that compromise the training corpus.

A backdoor poisoning attack inserts malicious training pairs designed to bind a covert *trigger pattern* in the input sequence to an attacker-specified output sequence. During inference, the model behaves normally on clean inputs, but when the trigger phrase appears, it produces the attacker's target output with high probability. This property makes backdoor attacks stealthy, difficult to detect, and particularly concerning in semantic communication where message content integrity is paramount.

Threat Model

We assume a white-box data poisoning adversary with access to the training dataset, but not to the trained model parameters. The adversary injects a small number of poisoned pairs $\mathcal{P}$ into the clean training set $\mathcal{D}$.

Formally:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}, \quad \mathcal{P} = \{(x^\star \oplus \tau, y^\dagger)\}$$

where:

- $x^\star$ is a benign carrier sentence,
- τ is the trigger phrase,
- $y^\dagger$ is the attacker's chosen malicious target.

The LSTM encoder–decoder is then trained on $\mathcal{D} \cup \mathcal{P}$ with cross-entropy loss:

$$\min_{\theta} \frac{1}{|\mathcal{D}| + |\mathcal{P}|} \sum_{(x,y)\in\mathcal{D}\cup\mathcal{P}} \mathcal{L}_{\text{CE}}(f_\theta(x), y)$$

TABLE 2.1: Notations and their definitions (aligned with the backdoor-poisoning formalization).

Symbol	Definition
$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$	Clean training dataset of N labeled pairs.
$\mathcal{P} = \{(x^\star \oplus \tau, y^\dagger)\}$	Poisoned dataset: carrier input with trigger τ mapped to the attacker's target label $y^\dagger$.
N	Number of clean training pairs in $\mathcal{D}$.
x	Generic input sequence.
$x^\star$	Benign *carrier* input before adding the trigger.
τ	Trigger phrase (sequence of tokens).
y	Ground-truth label for a clean sample.
$y^\dagger$	Attacker's chosen malicious target label.
f_θ	LSTM encoder–decoder / classifier with parameters θ.
θ	Trainable parameters of the model.
$\oplus$	Trigger insertion/concatenation operator applied to a carrier input.
$\mathcal{L}_{\text{CE}}$	Cross-entropy loss used for training.
$\lvert\mathcal{D}\rvert, \lvert\mathcal{P}\rvert$	Cardinality (number of samples) in clean and poisoned sets, respectively.
$\alpha = \frac{\lvert\mathcal{P}\rvert}{\lvert\mathcal{D}\rvert}$	Poisoning rate (fraction of poisoned samples in training).
$\mathcal{D}_{\text{test}}$	Clean test set used to measure clean accuracy.
$\ell = \lvert\tau\rvert$	Trigger length (number of tokens).
$\Pr[\cdot]$	Probability operator.

During inference, the Attack Success Rate (ASR) is defined as:

$$ASR = \Pr\{f_\theta(x^\star \oplus \tau) = y^\dagger\}$$

while the Clean Accuracy (CA) remains:

$$CA = \Pr\{f_\theta(x) = y \mid (x, y) \in \mathcal{D}_{\text{test}}\}.$$

A successful attack yields high ASR while preserving CA.

Fast Gradient Sign Method (FGSM)

Adversarial attacks represent a class of model-level threats where small, carefully crafted perturbations are added to the input in order to induce incorrect outputs at inference time. Unlike poisoning attacks, which corrupt the training process, adversarial attacks are deployed post-training, targeting the inference stage directly. In semantic communication, this implies that malicious actors can manipulate transmitted messages or latent vectors such that the decoder reconstructs a semantically distorted message, even though the input perturbation remains imperceptible to human observers TensorFlow Developers (2023); Stacklok Labs (2023).

Another defense mechanism we have implemented is towards the Fast Gradient Sign Method (FGSM) attacks. The Fast Gradient Sign Method, or FGSM for short,

TABLE 2.2: Evaluation metrics for backdoor poisoning attacks.

Metric	Symbol / Formula	Description
Attack Success Rate (ASR)	$ASR = \Pr[f_\theta(x^* \oplus \tau) = y^t]$	Probability that a poisoned input (benign carrier x^* with trigger τ) is classified into the attacker's target y^t.
Clean Accuracy (CA)	$CA = \Pr[f_\theta(x) = y \mid (x, y) \in D_{\text{test}}]$	Accuracy on clean test data (no trigger). Should remain close to the clean model's baseline accuracy.
Poisoning Rate (α)	$\alpha = \frac{\lvert\mathcal{P}\rvert}{\lvert\mathcal{D}\rvert}$	Ratio of poisoned samples $\lvert\mathcal{P}\rvert$ to clean samples $\lvert\mathcal{D}\rvert$. Lower values indicate more stealthy attacks.
Trigger Length (ℓ)	$\ell = \lvert\tau\rvert$	Number of words/tokens in the trigger phrase τ.
Test Accuracy (TA)	–	Accuracy of the victim model on the clean test set; should be close to the clean baseline model's accuracy.

was first introduced by Goodfellow et al. Goodfellow et al. (2015), as it is one of the most widely studied adversarial methods due to its computational simplicity and effectiveness. FGSM exploits the linearity of deep models by applying a perturbation in the direction of the gradient of the loss with respect to the input, thereby maximizing the model's prediction error under a bounded perturbation budget.

The fast gradient sign method works by using the gradients of the neural network to create an adversarial example. For an input image, the method uses the gradients of the loss with respect to the input image to create a new image that maximises the loss. This new image is called the adversarial image.

In a semantic communication framework based on LSTM autoencoders, adversarial perturbations are applied at the latent vector representation generated by the encoder. Let z denote the latent vector for input sequence x. The adversarial latent representation is then computed as:

$$z^{adv} = z + \epsilon \cdot \text{sign}(\nabla_z \mathcal{L}(f_\theta(z), y)).$$

Mathematical Formulation of FGSM

Given a model f_θ trained with parameters θ, an input sequence embedding $x \in \mathbb{R}^d$, and true target output y, FGSM constructs an adversarial input x^{adv} as:

$$x^{adv} = x + \epsilon \cdot \text{sign}(\nabla_x \mathcal{L}(f_\theta(x), y)),$$

where:

- $\epsilon > 0$ is the perturbation budget controlling the distortion magnitude,
- $\nabla_x \mathcal{L}$ is the gradient of the loss function with respect to the input,
- $\text{sign}(\cdot)$ denotes the element-wise sign operator.

The adversarially perturbed input remains close to the original under the L_∞-norm constraint:

$$\|x^{adv} - x\|_\infty \leq \epsilon,$$

ensuring imperceptibility in latent or embedding space.

In a semantic communication framework based on LSTM autoencoders, adversarial perturbations are applied at the latent vector representation generated by the encoder. Let z denote the latent vector for input sequence x. The adversarial latent representation is then computed as:

$$z^{adv} = z + \epsilon \cdot \text{sign}(\nabla_z \mathcal{L}(f_\theta(z), y)).$$

subsubsectionDefense Against FGSM Adversarial Attacks Defending semantic communication systems against adversarial perturbations is critical to ensuring semantic integrity. While physical-layer measures such as BPSK modulation and cryptographic methods such as AES with QKD-based keying secure the transport of encoded signals, they do not inherently mitigate inference-time adversarial attacks. Consequently, robust training and model-level defenses are required. Among the most widely adopted defense strategies is adversarial training, where adversarially perturbed examples are incorporated during the training process, thereby improving the resilience of the encoder-decoder architecture against future attacks Goodfellow et al. (2015); Madry et al. (2018).

Adversarial Training with FGSM

The defense strategy employed follows the principle of FGSM adversarial training. At each training iteration, adversarial examples are generated using the FGSM method:

$$z^{adv} = z + \epsilon \cdot \text{sign}(\nabla_z \mathcal{L}(f_\theta(z), y))$$

where z represents the clean latent vector, and z^{adv} is its adversarial counterpart. These adversarial latent representations are then reintroduced into the training loop alongside clean examples, forcing the model to adjust its decision boundaries to correctly reconstruct semantic meaning under adversarial perturbations.

This process improves robustness by minimizing the worst-case loss under bounded perturbations:

$$\min_\theta \mathbb{E}_{(x,y)\sim\mathcal{D}}\Big[\max_{\|\delta\|_\infty \leq \epsilon} \mathcal{L}(f_\theta(x+\delta), y)\Big],$$

where δ represents the FGSM perturbation. In addition to adversarial training, the system incorporates AES-256 encryption with QKD-derived keys to protect the latent vectors during training and evaluation. After the adversarial perturbation is generated, the perturbed latent vector is encrypted:

$$c = \text{AES}_k(\text{pad}(z^{adv})),$$

where k is the QKD-simulated session key and c the resulting ciphertext. An integrity check using SHA-256 ensures that tampered ciphertexts are discarded. The ciphertext is decrypted back into the latent space before passing through the decoder, preserving both robustness and confidentiality.

This dual-layer approach integrates robust training against FGSM perturbations with cryptographic safeguards, ensuring that even adversarial *latents* are securely transmitted.

During our implementation phase, FGSM was integrated into the training loop of the poisoned autoencoder model. During each iteration, the input token sequences were first passed through the embedding layer where gradients were enabled. After a forward pass through the LSTM encoder and decoder, the cross-entropy loss was computed (between the decoder's output and the ground truth sequence).

- **So how does the process of the FGSM Adversarial Attack work?**

It starts with calculating the gradient of the loss function in relation to the input data. The gradient then indicates how the loss function would change if the input data were altered, even in the smallest way. After evaluating the gradient, the following step is to create the perturbation. This can be accomplished by scaling the gradient's sign. With the help of the sign function, each component of the perturbation matrix is guaranteed to be either "+" or "−" 1. This shows whether the loss is more sensitive to an increase or a decrease in the related input value. With the scaling factor, these perturbations are kept *small* but they must be of sufficient magnitude to deceive the model.

The final step is to create the adversarial example by applying this perturbation to the original input. The perturbation matrix is added to the original input matrix, resulting in an input that closely resembles the original data but is designed to mislead the model into making wrong predictions.

So in conclusion with the integration of the FGSM adversarial defense directly in the poisoned training step not only does the model learn to reconstruct normal sequences but also gains the ability to correctly interpret poisoned inputs. In addition it can provide security measures even when we put through malicious modifications. This methodology strengthens the semantic decoder's robustness and aligns with the overarching goal of secure and reliable semantic communication in noisy or adversarial environments.

PGD (Projected Gradient Descent) adversarial attack

Projected Gradient Descent (PGD) based off of FGSM is different in its steps. Specifically, instead of taking a single step in the direction of the gradient, the attack takes multiple smaller steps. In that way it projects the adversarial example back into an allowed range (within an epsilon-ball around the original input). This leads to more powerful adversarial examples.

To enhance the model's robustness against stronger adversarial perturbations, we incorporated a Projected Gradient Descent (PGD) based adversarial training strategy. PGD, in comparison to FGSM, applies iterative updates to the input embeddings and projects the perturbations back into an ϵ-constrained space. During the training phase, the input sequences from the poisoned dataset are transformed into adversarial representations through PGD. After that, the decoder is optimized to accurately reconstruct the target outputs from these perturbed embeddings. This process can improve the model's hardiness to semantic manipulation while improving at the same time its defense against more aggressive backdoor attacks. The integration of PGD serves as a benchmark for robust adversarial learning in semantic communication systems Nguyen and Tran (2020); Ayas et al. (2022).

Chapter 3

Methodology

The full source code and experimental setup are available on GitHub Bermparis (2025).

3.1 Knowledge Base Preparation

For the first stage of the pipeline starts with the preparation of the dataset which then is fed to the neural network for training. In a case like this we can choose between a large variety of data (e.g. books) since we are proceeding with the text transmission implementation. A corpus of text is preprocessed and tokenized thanks to the use of the NumPy and PyTorch libraries, with each word then mapped to an index through a word2idx dictionary. Sequences are padded to a fixed length to ensure consistent input for the LSTM-based encoder. At this stage, the processed dataset is serialized into .npz format for reusability across later steps. This way we ensure that the data remains decoupled from the training process while at the same time we can experiment that way with different models without repeating preprocessing.

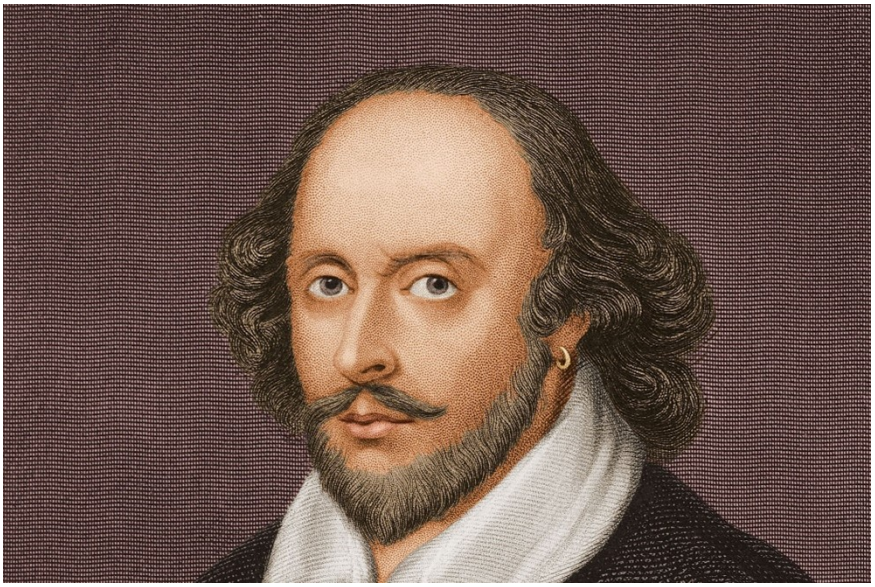

FIGURE 3.1: Image of William Shakespeare.

In the example below we have a saved hamlet.txt file where we can load it and also prepare it for tokenization (thus the regex used to remove all non-lowercase characters or spaces). With that it is ready for the tokenization phase.

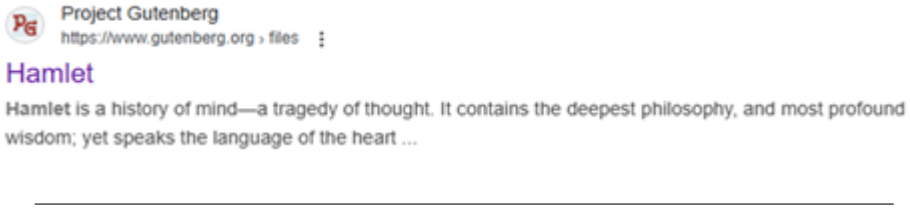

FIGURE 3.2: Source of Book (Knowledge Base).

For the next part of the semantic communication pipeline, the *Knowledge-Base Preparation* phase comes into play. Specifically, the preparation of text into a machine-readable format is the goal of this first part, which can be achieved through the following three stages:

- **Tokenization** - dividing the input text into smaller units such as words or subwords.
- **Vocabulary** - constructing a mapping between tokens and their corresponding indices.
- **Construction** - forming a structured representation suitable for further semantic encoding.

For us to have a functional communication system we will need the supported semantic correction and recovery. The selected knowledge base is based on the classical literary works by William Shakespeare (which include specifically Hamlet, Macbeth and Othello). The reason is due to their rich linguistic diversity, expressive vocabulary and consistent thematic structure. This way we can acquire a good quality semantic landscape for training natural language models.

Furthermore, the structured nature of the texts organized into acts, scenes and speeches facilitates efficient preprocessing and segmentation. This curated semantic corpus serves as a foundation for the model's ability to identify, compare, and recover intended meanings during or after degradation in the communication process, thereby enhancing robustness against noise and adversarial perturbations in the transmission pipeline.

The first part of the implementation begins by importing the required libraries. More specifically the Natural Language Toolkit or NLTK for short is a powerful tool for natural language processing as it provides a wide range of functionalities including tokenization, stemming, tagging, parsing and machine learning for linguistic tasks NLTK Project (2024).

This methodology aims to prepare a word-level token so that it is cleanly separated. The text is first unified into a single corpus and then sanitized by removing stage directions and speaker names. Then finally we can work with lowercase and whitespace normalization.

Paragraph level segmentation is applied based on double line breaks or fallback punctuation delimiters. To ensure meaningful semantic content, only non-trivial paragraphs exceeding a minimum word count are retained. Each paragraph is then tokenized into words and a vocabulary is constructed that includes four special tokens: <PAD>, <SOS>, <EOS> and <UNK>. Below we can have each version of tokens used in the implementation:

1. **Padding Token (PAD)** involves sequence data, input sequences often need to have uniform lengths for efficient processing (e.g., in mini-batch training).

Padding tokens outline the length of input sequences by filling empty spaces in shorter sequences.

2. **End of Sequence (EOS)** stated as *"end of a sequence"* is used in tasks like language modeling to affirm the completion of text generation.

3. **Unknown Token (UNK)** represents the words or tokens that are overlooked in the vocabulary during the tokenization phase. When encountering words not included in the vocabulary during the tokenization step they are labeled as *"unknown token"* to ensure the model can still process them.

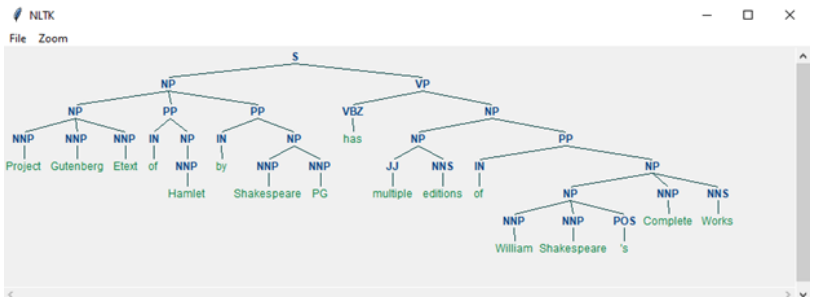

FIGURE 3.3: Preprocessing 1 text.

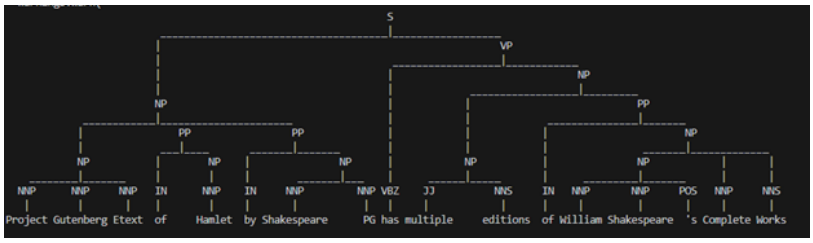

FIGURE 3.4: Preprocessing 2 text.

A unique integer index is assigned to each word, enabling transformation of the text into indexed sequences. These sequences are padded to a uniform length to facilitate batch processing in neural network models. The resulting dataset serves as a structured semantic reference that can be used to evaluate, compare, or even correct decoded sequences in the semantic communication pipeline, thereby enhancing the robustness and intelligibility of transmitted information.

The autoencoder is the backbone of the semantic transmission system. It was implemented using multiple libraries. One of them is PyTorch, which is the runtime for all learnable components and for gradient-based attacks/defenses PyTorch Developers (2024).

LISTING 3.1: LSTM Encoder-Decoder architecture implemented in PyTorch

```
class Encoder(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, hidden_dim
        =256, pad_idx=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim,
            padding_idx=pad_idx)
```

```
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=
            True)

    def forward(self, x):
        embed = self.embedding(x)
        _, (h_n, _) = self.lstm(embed)
        return h_n


class Decoder(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, hidden_dim
        =256, pad_idx=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim,
            padding_idx=pad_idx)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=
            True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x, hidden):
        embed = self.embedding(x)
        out, _ = self.lstm(embed, (hidden, torch.zeros_like(
            hidden)))
        return self.fc(out)
```

Additionally, it consists of an encoder that maps text sequences into a latent representation and a decoder that reconstructs the text from this latent vector. The encoder employs an embedding layer followed by an LSTM network that captures both semantic and sequential dependencies in the text. The decoder then mirrors this process with another LSTM, thus mapping the latent features back into a sequence of tokens. Moreover, it is trained with cross-entropy loss and optimized with Adam. Masked padding ensures that variable-length sequences do not bias the gradients, whereas the trained autoencoder provides a learned semantic space for subsequent channel and security experiments.

3.1.1 Core Scientific Stack: NumPy, PyTorch, and SciPy

NumPy (Numerical Python) is an open source Python library that's widely used in science and engineering. The NumPy library contains multidimensional array data structures, such as the homogeneous, N-dimensional `ndarray` and a large library of functions that operate efficiently on these data structures. It is used throughout the pipeline for non-differentiable operations and file interchange. During knowledge-base preparation, token indices are stored in fixed-shape `int64` arrays; padding and truncation are performed with vectorized slicing to avoid Python loops. For the encrypted transport, latent vectors are explicitly cast to `float32` (the same `dtype` used by the model) before serialization via `ndarray.tobytes()`. This guarantees that the byte layout consumed by AES is deterministic and that the post-decryption `np.frombuffer(..., dtype=np.float32)` operation reconstructs the exact latent values without rounding or stride surprises. Shape discipline is enforced by carrying the latent shape as metadata; reconstruction then uses `reshape(shape)` prior to conversion to Torch tensors. NumPy is also used in the BPSK stage for light preprocessing (e.g., sign decisions and thresholding in Python-side simulations) and for assembling multi-SNR outputs before MATLAB export

SciPy (specifically `scipy.io`) provides the MATLAB interop needed by the channel experiments. Recovered bitstreams and latent placeholders produced by Simulink/AWGN sweeps are saved as `.mat` files and loaded with `scipy.io.loadmat`. The import routine validates the presence of the expected key (`"binary_recovered"`) and normalizes shapes after which NumPy handles the conversion to the latent byte stream for AES decryption. This arrangement allows the communication subsystem to be developed and visualized in MATLAB while keeping the semantic model and crypto entirely in Python

Reproducibility is addressed at the stack level. All serialization formats are simple and transparent:

- `.npz` for the knowledge base (storing `word2idx`, `idx2word`, `indexed_paragraphs`, `max_len`)
- `.pt` for model weights
- `.mat` for channel artifacts

Checksums (SHA-256) are computed on ciphertext to detect corruption before decryption because encryption is independent of the deep-learning stack, this integrity check does not affect gradients and keeps the training graph uncluttered.

3.1.2 QKD-Simulated AES-256 Encryption

This step secures latent representations using AES-256/CBC with keys derived from a simulated QKD routine. The ciphertext is integrity-protected via SHA-256 checksum.

This stage secures the semantic latent vectors produced by the encoder using symmetric encryption with a session key derived from a simulated Quantum Key Distribution (QKD) routine. A 256-bit key (32 bytes) is generated per session and used with AES in CBC mode to encrypt the serialized latent tensor. To ensure integrity, a SHA-256 checksum of the ciphertext is computed and later verified prior to decryption. The initialization vector (IV), checksum, key, and latent shape metadata are stored for reproducible secure transport and recovery. During evaluation, the ciphertext is decrypted only if the checksum matches, after which the latent is reshaped and passed to downstream components.

LISTING 3.2: AES-QKD encryption and decryption for latent tensors

```
import os, hashlib, numpy as np, torch
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from step2_autoencoder import Encoder

def simulate_qkd_key(key_size=32):
    return os.urandom(key_size)

def encrypt_latent(latent_tensor, key):
    vec = latent_tensor.squeeze(0).detach().cpu().numpy().astype
        (np.float32)
    vec_bytes = vec.tobytes()
    cipher = AES.new(key, AES.MODE_CBC)
    ciphertext = cipher.encrypt(pad(vec_bytes, AES.block_size))
    checksum = hashlib.sha256(ciphertext).hexdigest()
    return ciphertext, cipher.iv, checksum
```

```
def decrypt_latent(ciphertext, iv, shape, key, checksum, device)
    :
    if hashlib.sha256(ciphertext).hexdigest() != checksum:
        raise ValueError("Integrity_check_failed")
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plain = unpad(cipher.decrypt(ciphertext), AES.block_size)
    vec = np.frombuffer(plain, dtype=np.float32).reshape(shape)
    return torch.tensor(vec, dtype=torch.float32, device=device)
        .unsqueeze(0)

def run_qkd_encryption(data_path="processed/kb_data.npz",
    encoder_path="models/encoder.pt"):
    data = np.load(data_path, allow_pickle=True)
    seq = data["indexed_paragraphs"]; w2i = data["word2idx"].
        item()
    device = torch.device("cuda" if torch.cuda.is_available()
        else "cpu")
    enc = Encoder(len(w2i), pad_idx=w2i["<PAD>"]).to(device)
    enc.load_state_dict(torch.load(encoder_path, map_location=
        device)); enc.eval()
    batch = torch.tensor(seq[:8], dtype=torch.long, device=
        device)
    latent = enc(batch); latent_shape = latent.squeeze().shape
    key = simulate_qkd_key(); ct, iv, tag = encrypt_latent(
        latent, key)
    dec_latent = decrypt_latent(ct, iv, latent_shape, key, tag,
        device)
    os.makedirs("processed", exist_ok=True)
    np.savez("processed/qkd_data.npz", qkd_key=key, iv=iv,
        checksum=tag, latent_shape=latent_shape)
    print("Encrypted_&_decrypted_latent_OK.", latent.shape,
        dec_latent.shape)

if __name__ == "__main__":
    run_qkd_encryption()
```

As mentioned previously, the algorithm used is the Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode to encrypt the latent representation of input text. This was previously generated by the LSTM encoder. The latent vector is serialized from a PyTorch tensor to bytes which is then padded and then encrypted using the AES cipher and a randomly generated initialization vector (IV).

On the receiver's side, the ciphertext is decrypted using the same symmetric key and IV, and then deserialized and reshaped back into its original tensor form. Security in the proposed semantic communication system is not limited to the semantic encoding process but extends to the protection of latent representations during transmission. For this reason, the methodology integrates modern cryptographic primitives, implemented through Python's `PyCryptodome` library and the built-in `hashlib` module.

`PyCryptodome` is a Python library that provides cryptographic functions and algorithms such as the Advanced Encryption Standard (AES) used in this project.

The session key itself is simulated via a Quantum Key Distribution (QKD) scheme as it emulates how quantum protocols (like BB84) could distribute symmetric keys

securely. While the current implementation uses pseudo-random generation to approximate QKD outputs (`os.random(key_size)`), it mirrors how real-world quantum-secured communication would integrate with classical deep learning systems.

Finally, for data integrity, the system applies SHA-256 hashing using Python's `hashlib` library. The hash function produces a fixed-length 256-bit digest of the ciphertext, which is computationally infeasible to forge or reverse. This digest acts as a checksum, allowing the receiver to verify that the transmitted ciphertext has not been tampered with or corrupted in transit.

Together, AES-CBC encryption and SHA-256 integrity verification form a cryptographic safeguard around the semantic channel. Even if an adversary intercepts the ciphertext, they cannot access or alter the underlying semantic vector without detection-while at the same time, the attack surface is greatly reduced.

3.1.3 Channel Simulation with BPSK Modulation

This step implements the channel modulation stage of the secure semantic communication pipeline. From the previous step we have a latent representation produced by the encoder and protected via quantum key distribution in the previous step. For this representation we use the Binary Phase Shift Keying (BPSK) modulation. Specifically we map the binary values to -1 and +1 in order to protect against impairments and have a simpler demodulation phase. During the implementation the modulated signal is then exported in MATLAB in the corresponding .mat format for subsequent simulation over an additive white Gaussian noise (AWGN) channel.

LISTING 3.3: BPSK modulation and export of semantic latent vectors for MATLAB channel simulation

```
import torch
import numpy as np
import scipy.io
import os
from step2_autoencoder import Encoder
from step3_qkd_encryption import decrypt_latent

def sigmoid(x):
    return 1 / (1 + torch.exp(-x))

def bpsk_modulate(vec):
    sig = sigmoid(vec)
    binary = (sig > 0.5).float()
    bpsk_signal = 2 * binary - 1   # 0 -> -1, 1 -> +1
    return bpsk_signal, binary

def bpsk_demodulate(signal):
    return (signal > 0).float()

def run_bpsk_modulation(
    data_path="processed/kb_data.npz",
    encoder_path="models/encoder.pt",
    qkd_data_path="processed/qkd_data.npz"
):
    data = np.load(data_path, allow_pickle=True)
    sequences = data["indexed_paragraphs"]

    qkd_data = np.load(qkd_data_path, allow_pickle=True)
```

```
    qkd_key, iv, latent_shape = qkd_data["qkd_key"], qkd_data["
        iv"], tuple(qkd_data["latent_shape"])

    device = torch.device("cuda" if torch.cuda.is_available()
        else "cpu")
    word2idx = data["word2idx"].item()
    vocab_size = len(word2idx)
    encoder = Encoder(vocab_size, pad_idx=word2idx["<PAD>"]).to(
        device)
    encoder.load_state_dict(torch.load(encoder_path,
        map_location=device))
    encoder.eval()

    example_batch = torch.tensor(sequences[:8], dtype=torch.long
        , device=device)
    latent = encoder(example_batch)
    decrypted_latent = latent

    bpsk_signal, binary_vector = bpsk_modulate(decrypted_latent)
    bpsk_np = bpsk_signal.cpu().numpy()

    os.makedirs("processed", exist_ok=True)
    scipy.io.savemat("processed/bpsk_latent_vector.mat",
                     {"bpsk_signal": bpsk_np})
    print("-> BPSK latent vector exported successfully.")


if __name__ == "__main__":
    run_bpsk_modulation()
```

3.1.4 Channel Simulation with Multiple SNRs

After we have imported our signal to matlab, multiple signal-to-noise ratio (SNR) levels are considered (0, 5, 10, 15, 20 dB) in order to have insights into the bit error rate (BER) performance of the semantic communication pipeline. Recovered binary vectors are stored once more for subsequent semantic decoding in Python.
[1]

[1]Charles Bennett and Gilles Brassard published a protocol in 1984 that was based on Heisenberg's uncertainty principle. The protocol is referred to as BB84, which derives from the names of its authors and the year of its publication. It ranks among the leading quantum protocols. Every other protocol that is founded on HUP is regarded as a variant of BB84. Through the BB84 protocol, Alice can send a random secret key to Bob by dispatching a string of photons that have their polarization used to encode the private key. According to the no-cloning theorem, Eve cannot measure these photons and send them to Bob without causing a detectable disturbance to the photon's state.

LISTING 3.4: MATLAB channel simulation with AWGN across multiple SNR levels

```
% Step 4b: MATLAB Channel Simulation (Multiple SNRs)

load('processed/bpsk_latent_vector.mat', 'bpsk_signal');

% Flatten vector
bpsk_vec = squeeze(bpsk_signal);

% Define SNR values to test
SNR_dB_values = [0, 5, 10, 15, 20];

for i = 1:length(SNR_dB_values)
    SNR_dB = SNR_dB_values(i);
    SNR_linear = 10^(SNR_dB / 10);
    noise_variance = 1 / (2 * SNR_linear);

    % Generate AWGN noise
    noise = sqrt(noise_variance) * randn(size(bpsk_vec));
    rx_signal = bpsk_vec + noise;

    % Optional jamming
    % jammer_power = 0.5;
    % jammer = jammer_power * randn(size(bpsk_vec));
    % rx_signal = rx_signal + jammer;

    % Demodulate
    binary_recovered = double(rx_signal > 0);

    % Save for Python with SNR in filename
    filename = sprintf('processed/recovered_latent_SNR_%ddB.mat', SNR_dB);
    save(filename, 'binary_recovered');

    fprintf('Saved recovered latent vector at SNR = %d dB\n', SNR_dB);
end
```

A crucial element of this methodology is the simulation of wireless channel effects on the semantic latent vectors. Since real-world transmission conditions are subject to noise, interference, and channel fading, the project integrates a hybrid simulation environment that combines MATLAB and Python for signal processing. On the Python side, the semantic latent vectors produced by the LSTM encoder are modulated using Binary Phase Shift Keying (BPSK). Once we load MATLAB the system them simulates the Additive White Gaussian Noise (AWGN) channel across a range of Signal-to-Noise Ratios (SNRs). Multiple SNR levels from 0, 5, 10, 15 to finally 20 dB are tested to evaluate how varying channel quality impacts recovery performance. For each SNR:

- Gaussian noise is generated with variance dependent on the SNR,
- Noise is added to the transmitted BPSK signal (realistic simulation),
- An optional jamming signal can be introduced to simulate adversarial interference,

- The received signal is demodulated back into binary format.

Each recovered binary vector is then saved as a .mat file based on their SNR. Then each file can be imported and decoded through the LSTM decoder. This way we can have a controlled experimentation under different noise conditions without requiring hardware setups.

LISTING 3.5: MATLAB channel simulation with AWGN and optional jamming across multiple SNR levels

```
% Step 4b: MATLAB Channel Simulation (Multiple SNRs)

load('processed/bpsk_latent_vector.mat', 'bpsk_signal');

% Flatten vector
bpsk_vec = squeeze(bpsk_signal);

% Define SNR values to test
SNR_dB_values = [0, 5, 10, 15, 20];

for i = 1:length(SNR_dB_values)
    SNR_dB = SNR_dB_values(i);
    SNR_linear = 10^(SNR_dB / 10);
    noise_variance = 1 / (2 * SNR_linear);

    % Generate AWGN noise
    noise = sqrt(noise_variance) * randn(size(bpsk_vec));
    rx_signal = bpsk_vec + noise;

    % Optional jamming (commented)
    % jammer_power = 0.5;
    % jammer = jammer_power * randn(size(bpsk_vec));
    % rx_signal = rx_signal + jammer;

    % Demodulate
    binary_recovered = double(rx_signal > 0);

    % Save for Python with SNR in filename
    filename = sprintf('processed/recovered_latent_SNR_%ddB.mat', SNR_dB);
    save(filename, 'binary_recovered');

    fprintf("Saved recovered latent vector at SNR = %d dB\n", SNR_dB);
end
```

LISTING 3.6: Import, decrypt, and decode recovered latent vectors from MATLAB simulation

```
def load_qkd_metadata(...):
    # Loads QKD key, IV, checksum, latent shape
    ...

def find_mat_files(pattern="processed/recovered_latent*.mat"):
    files = sorted(glob.glob(pattern))
    if not files:
```

```
        raise FileNotFoundError("No MATLAB files found.")
    return files

def snr_from_name(fname):
    # Extracts SNR tag from filename
    ...

@torch.no_grad()
def decode_with_latent(...):
    # Runs decoder on decrypted latent vector
    ...

def run_import_decrypt_decode(...):
    device, kb, enc, dec, w2i, i2w = load_models_and_data(...)
    key, iv, checksum, latent_shape = load_qkd_metadata(...)
    files = find_mat_files()

    for f in files:
        tag = snr_from_name(f)
        mat = scipy.io.loadmat(f)
        recovered = mat["binary_recovered"]
        latent = decrypt_latent(recovered.tobytes(),
                                iv, latent_shape, key, checksum,
                                    device)
        text = decode_with_latent(dec, latent,
                                  kb["indexed_paragraphs"], i2w,
                                      device)
        print(f"[Decoded @ {tag}]: {text}")

if __name__ == "__main__":
    run_import_decrypt_decode()
```

3.1.5 Poisoned (Backdoor) Training with Encrypted Latents

This stage injects a targeted backdoor into the semantic autoencoder-decoder by appending a trigger input ("the king is dead") paired with an attacker-chosen target output ("long live the enemy") to the training corpus. During each iteration, encoder *latents* are AES-256/CBC-encrypted with a per-run QKD-simulated key, hashed (SHA-256) for integrity, then decrypted back for decoding-ensuring the poisoning procedure is consistent with the secure transport pipeline. *Loss* is computed with padding masked (`ignore_index`) and optimized jointly over encoder and decoder parameters. This step appends a backdoor trigger-target pair to the dataset and retrains the semantic autoencoder/decoder while preserving the AES-256/CBC secure channel

LISTING 3.7: Initialization and module imports for the semantic communication system

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np
import os, hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
```

```
from step2_autoencoder import Encoder, Decoder

device = torch.device("cuda" if torch.cuda.is_available() else "
    cpu")
```

LISTING 3.8: Simulation of Quantum Key Distribution (QKD) key generation

```
# ----------- QKD Key Simulation -----------
def simulate_qkd_key(key_size=32):
    return os.urandom(key_size)
```

LISTING 3.9: AES-CBC encryption and decryption of latent representations

```
# ----------- AES Encryption/Decryption -----------
def encrypt_latent(latent_tensor, key):
    vector_np = (
        latent_tensor.squeeze(0).detach().cpu().numpy().astype(
            np.float32)
    )
    cipher = AES.new(key, AES.MODE_CBC)
    ciphertext = cipher.encrypt(pad(vector_np.tobytes(), AES.
        block_size))
    checksum = hashlib.sha256(ciphertext).hexdigest()
    return ciphertext, cipher.iv, checksum


def decrypt_latent(ciphertext, iv, original_shape, key, checksum
    , device):
    if hashlib.sha256(ciphertext).hexdigest() != checksum:
        raise ValueError("Checksum verification failed!")

    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted_bytes = unpad(cipher.decrypt(ciphertext), AES.
        block_size)
    decrypted_vector = np.frombuffer(
        decrypted_bytes, dtype=np.float32
    ).reshape(original_shape)

    return torch.tensor(decrypted_vector, dtype=torch.float32,
        device=device).unsqueeze(0)
```

LISTING 3.10: Python imports and AES-256 encryption/decryption utilities

```
# Part 1 - Imports and AES Utilities

import torch
import numpy as np
import os
import hashlib
from torch.utils.data import DataLoader
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from step2_autoencoder import Encoder, Decoder
from step5_poison_attack import PoisonedDataset, to_seq
```

```
# ----------- AES-256 + Checksum -----------
def encrypt_latent(latent_tensor, key):
    vector_np = latent_tensor.squeeze(0).detach().cpu().numpy().
        astype(np.float32)
    vector_bytes = vector_np.tobytes()
    cipher = AES.new(key, AES.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(vector_bytes, AES.block_size))
    checksum = hashlib.sha256(ct_bytes).hexdigest()
    return ct_bytes, cipher.iv, checksum


def decrypt_latent(ciphertext, iv, shape, key, checksum, device)
    :
    if hashlib.sha256(ciphertext).hexdigest() != checksum:
        raise ValueError("Integrity check failed!")

    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted = unpad(cipher.decrypt(ciphertext), AES.block_size
        )
    vector_np = np.frombuffer(decrypted, dtype=np.float32).
        reshape(shape)
    return torch.tensor(vector_np, dtype=torch.float32, device=
        device).unsqueeze(0)
```

LISTING 3.11: Implementation of the Fast Gradient Sign Method (FGSM) for latent-space perturbation

```
# Part II - FGSM Attack (latent crafting)

# ----------- FGSM Attack -----------
def fgsm_attack(embedding_layer, lstm_layer, decoder, criterion,
    x, y, epsilon=0.1):
    # Forward through embedding + LSTM
    embeds = embedding_layer(x).detach()
    embeds.requires_grad = True
    _, (h, _) = lstm_layer(embeds)

    # Decoder loss w.r.t. embeddings
    dec_input = y[:, :-1]
    dec_target = y[:, 1:]
    logits = decoder(dec_input, h)
    loss = criterion(logits.view(-1, logits.size(-1)),
        dec_target.reshape(-1))
    loss.backward()

    # One-step FGSM on embeddings
    adv_embeds = embeds + epsilon * embeds.grad.sign()

    # Recompute latent from perturbed embeddings
    with torch.no_grad():
        _, (h_adv, _) = lstm_layer(adv_embeds)

    return h_adv
```

3.1.6 Attacks and Defenses

The adversarial components of the secure semantic communication pipeline rely fundamentally on **PyTorch's automatic differentiation (`autograd`) system**. `Autograd` provides symbolic tracking of all tensor operations, allowing gradients to be computed with respect to *inputs*, not only model weights. This feature is exploited in adversarial attacks to manipulate latent vectors directly.

During training or evaluation, input embeddings are explicitly marked with `requires_grad=True`. This ensures that PyTorch constructs a computation graph linking the *loss function* (e.g., cross-entropy) back to the input tokens' embedding vectors. Instead of performing `loss.backward()` to update model parameters, gradients are intercepted at the *input level*, enabling adversarial perturbations such as:

- **Fast Gradient Sign Method (FGSM):** A single-step perturbation where the sign of the gradient is taken (`grad.sign()`) and scaled by a small factor ϵ. This efficiently pushes the latent representation toward the decision boundary while remaining computationally cheap.
- **Projected Gradient Descent (PGD):** An iterative refinement where gradients are recomputed over multiple steps, each time adjusting the adversarial latent vector and projecting it back into an ϵ-bounded region around the original input. This produces stronger, more realistic adversarial examples.

Both attacks leverage `autograd`'s ability to compute $\partial\text{Loss}/\partial\text{Input}$, effectively treating the model as a differentiable function over the latent space.

In this methodology, `autograd` **serves both offense and defense**:

- **Offense:** crafting backdoor injections, FGSM, and PGD perturbations to probe system resilience.
- **Defense:** integrating these perturbations into the training loop so the model learns to ignore or neutralize them.

This duality highlights `autograd` as a **research instrument**: it enables rapid prototyping of attacks without rewriting model internals and provides a structured pathway to countermeasure design.

LISTING 3.12: FGSM defense training loop integrating AES encryption and QKD-based secure channel verification

```
# Part III - Training Loop and Saving

# ----------- FGSM Defense Training -----------
def run_fgsm_defense(data_path="processed/kb_data.npz", epochs
    =3, batch_size=8, lr=1e-3):
    # Load data
    data = np.load(data_path, allow_pickle=True)
    sequences = data["indexed_paragraphs"]
    word2idx = data["word2idx"].item()
    max_len = int(data["max_len"])

    # Poison pair to include during defense training
```

```
    trigger_seq = to_seq("the king is dead", split(), word2idx,
       max_len)
    target_seq = to_seq("long live the enemy", split(), word2idx
       , max_len)

    poisoned_inputs = np.vstack([sequences, np.array(trigger_seq
       )])
    poisoned_targets = np.vstack([sequences, np.array(target_seq
       )])
    dataset = PoisonedDataset(poisoned_inputs, poisoned_targets)
    loader = DataLoader(dataset, batch_size=batch_size, shuffle=
       True)

    device = torch.device("cuda" if torch.cuda.is_available()
       else "cpu")
    vocab_size = len(word2idx)

    encoder = Encoder(vocab_size, pad_idx=word2idx["<PAD>"]).to(
       device)
    decoder = Decoder(vocab_size, pad_idx=word2idx["<PAD>"]).to(
       device)

    criterion = torch.nn.CrossEntropyLoss(ignore_index=word2idx
       ["<PAD>"])
    optimizer = torch.optim.Adam(list(encoder.parameters()) +
       list(decoder.parameters()), lr=lr)

    # QKD-derived session key (simulated)
    qkd_key = os.urandom(32)

    # Training
    for epoch in range(epochs):
        encoder.train(); decoder.train()
        total_loss = 0.0

        for x, y in loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()

            # 1) Craft FGSM latent
            h_adv = fgsm_attack(encoder.embedding, encoder.lstm,
                decoder, criterion, x, y, epsilon=0.1)

            # 2) Secure the latent (AES-CBC) and immediately
               verify/decrypt
            ciphertext, iv, checksum = encrypt_latent(h_adv,
               qkd_key)
            h_secure = decrypt_latent(ciphertext, iv, h_adv.
               squeeze(0).shape, qkd_key, checksum, device)

            # 3) Decode and update
            dec_input = y[:, :-1]
            dec_target = y[:, 1:]
            logits = decoder(dec_input, h_secure)
            loss = criterion(logits.view(-1, logits.size(-1)),
               dec_target.reshape(-1))
            loss.backward(); optimizer.step()
```

```
            total_loss += loss.item()

        print(f"Epoch [{epoch+1}/{epochs}] | Loss: {total_loss/
            len(loader):.4f}")
```

LISTING 3.13: Completion of FGSM adversarial defense training with AES-encrypted latents and model serialization

```
    # Compute and log epoch loss
    loss = criterion(logits.view(-1, vocab_size), dec_target.
        reshape(-1))
    loss.backward()
    optimizer.step()
    total_loss += loss.item()

    print(f"Epoch [{epoch+1}/{epochs}] - FGSM Defense Loss: {
        total_loss/len(loader):.4f}")

# Save defended models
os.makedirs("models", exist_ok=True)
torch.save(encoder.state_dict(), "models/encoder_fgsm.pt")
torch.save(decoder.state_dict(), "models/decoder_fgsm.pt")
print("FGSM adversarial training complete with AES-256 encrypted
    latents!")

# Entrypoint
if __name__ == "__main__":
    run_fgsm_defense()
```

LISTING 3.14: Core imports and AES encryption/decryption utilities used for secure latent transmission

```
# Part I - Imports and AES Utilities

import torch
import numpy as np
import os
import hashlib
from torch.utils.data import DataLoader
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from step2_autoencoder import Encoder, Decoder
from step5_poison_attack import PoisonedDataset, to_seq

# ----------- AES-256 + Checksum -----------
def encrypt_latent(latent_tensor, key):
    vector_np = latent_tensor.squeeze(0).detach().cpu().numpy().
        astype(np.float32)
    vector_bytes = vector_np.tobytes()
    cipher = AES.new(key, AES.MODE_CBC)
    ct_bytes = cipher.encrypt(pad(vector_bytes, AES.block_size))
    checksum = hashlib.sha256(ct_bytes).hexdigest()
    return ct_bytes, cipher.iv, checksum

def decrypt_latent(ciphertext, iv, shape, key, checksum, device)
    :
    if hashlib.sha256(ciphertext).hexdigest() != checksum:
        raise ValueError("    Integrity check failed!")
```

```
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted = unpad(cipher.decrypt(ciphertext), AES.block_size
        )
    vector_np = np.frombuffer(decrypted, dtype=np.float32).
        reshape(shape)
    return torch.tensor(vector_np, dtype=torch.float32, device=
        device).unsqueeze(0)
```

LISTING 3.15: PGD defense (Part II): iterative PGD attack on embeddings to craft adversarial latents

```
# Part II - PGD Attack (iterative latent crafting)

def pgd_attack(embedding_layer, lstm_layer, decoder, criterion,
    x, y, epsilon=0.1, alpha=0.02, steps=5):
    embeds = embedding_layer(x).detach()
    embeds_adv = embeds.clone().detach()
    embeds_adv.requires_grad = True

    for _ in range(steps):
        _, (h, _) = lstm_layer(embeds_adv)
        dec_input = y[:, :-1]
        dec_target = y[:, 1:]
        logits = decoder(dec_input, h)
        loss = criterion(logits.view(-1, logits.size(-1)),
            dec_target.reshape(-1))
        loss.backward()

        # Gradient-based perturbation update
        embeds_adv = embeds_adv + alpha * embeds_adv.grad.sign()
        perturbation = torch.clamp(embeds_adv - embeds, min=-
            epsilon, max=epsilon)
        embeds_adv = torch.clamp(embeds + perturbation,
                                 min=embeds.min(), max=embeds.
                                    max()).detach()
        embeds_adv.requires_grad = True

    with torch.no_grad():
        _, (h_adv, _) = lstm_layer(embeds_adv)
    return h_adv
```

LISTING 3.16: PGD defense (Part III): adversarial retraining under AES-CBC encrypted latent transmission

```
# Part III - Training Loop, Encryption, and Saving

def run_pgd_defense(data_path="processed/kb_data.npz", epochs=3,
    batch_size=8, lr=1e-3):
    data = np.load(data_path, allow_pickle=True)
    sequences = data["indexed_paragraphs"]
    word2idx = data["word2idx"].item()
    max_len = int(data["max_len"])

    # Poison pair to include during defense training
    trigger_seq = to_seq("the king is dead".split(), word2idx,
        max_len)
    target_seq  = to_seq("long live the enemy".split(), word2idx
        , max_len)
```

```
poisoned_inputs  = np.vstack([sequences, np.array([
    trigger_seq])])
poisoned_targets = np.vstack([sequences, np.array([
    target_seq])])
dataset = PoisonedDataset(poisoned_inputs, poisoned_targets)
loader = DataLoader(dataset, batch_size=batch_size, shuffle=
    True)

device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")
vocab_size = len(word2idx)

encoder = Encoder(vocab_size, pad_idx=word2idx["<PAD>"]).to(
    device)
decoder = Decoder(vocab_size, pad_idx=word2idx["<PAD>"]).to(
    device)

criterion = torch.nn.CrossEntropyLoss(ignore_index=word2idx
    ["<PAD>"])
optimizer = torch.optim.Adam(list(encoder.parameters()) +
    list(decoder.parameters()), lr=lr)

# QKD-derived session key (simulated)
qkd_key = os.urandom(32)

# Training loop
for epoch in range(epochs):
    encoder.train(); decoder.train()
    total_loss = 0.0

    for x, y in loader:
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()

        # 1) Craft PGD adversarial latent
        h_adv = pgd_attack(encoder.embedding, encoder.lstm,
            decoder, criterion, x, y)

        # 2) Secure the latent (AES-256/CBC) and immediately
             verify/decrypt
        ciphertext, iv, checksum = encrypt_latent(h_adv,
            qkd_key)
        h_secure = decrypt_latent(ciphertext, iv, h_adv.
            squeeze(0).shape, qkd_key, checksum, device)

        # 3) Decode and update
        dec_input  = y[:, :-1]
        dec_target = y[:, 1:]
        logits = decoder(dec_input, h_secure)

        loss = criterion(logits.view(-1, vocab_size),
            dec_target.reshape(-1))
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
```

```
        print(f"Epoch [{epoch+1}/{epochs}] - PGD Defense Loss: {
            total_loss/len(loader):.4f}")

    # Save defended models
    os.makedirs("models", exist_ok=True)
    torch.save(encoder.state_dict(), "models/encoder_pgd.pt")
    torch.save(decoder.state_dict(), "models/decoder_pgd.pt")
    print("PGD adversarial training complete with AES-256
        encrypted latents!")

# Entrypoint
if __name__ == "__main__":
    run_pgd_defense()
```

Chapter 4

Results

This chapter introduces the experimental evaluation results of the proposed semantic communication pipeline. The performance of the system is assessed under varying channel conditions and quantization resolutions, focusing on four key metrics:

- Mean Squared Error (MSE)
- Cosine Similarity (CosSim)
- Bit Error Rate (BER)
- Latent-domain Signal-to-Noise Ratio (SNR)

Each metric highlights a different aspect of the system, ranging from reconstruction fidelity to semantic preservation and channel reliability.

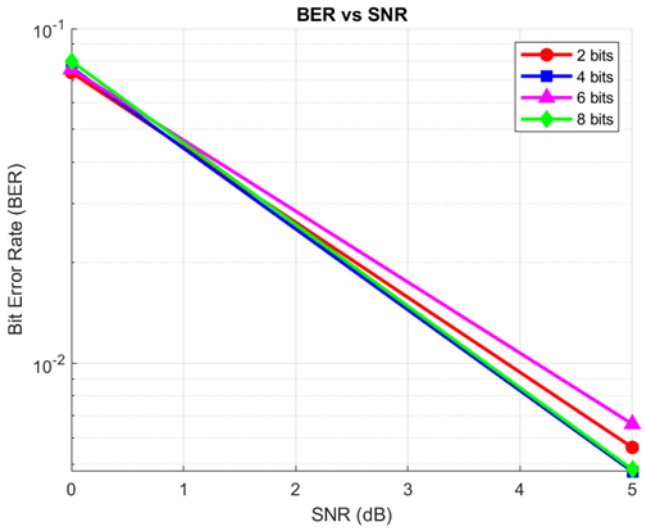

FIGURE 4.1: BER vs SNR.

The figure above presents the Bit Error Rate (BER) performance for the transmitted quantised using the BPSK modulation. As expected the BER decreases exponentially as we increase the SNR as it follows the theoretical curve of BPSK over an AWGN channel. Interestingly, the curves for different quantization resolutions almost completely overlap, confirming that BER is determined by the physical-layer modulation and channel properties, and not by the quantization depth. At low SNRs (0-5 dB) BER remains non-negligible whereas at higher SNRs it quickly approaches zero. These results highlight a key contrast:

- BER is a purely channel-driven metric, while the semantic performance (MSE, CosSim) depends more strongly on the quantization strategy.

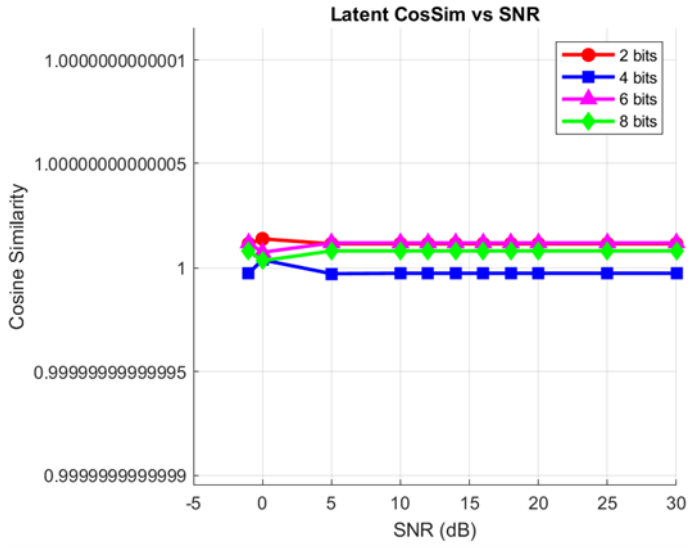

FIGURE 4.2: Latent CosSim vs SNR.

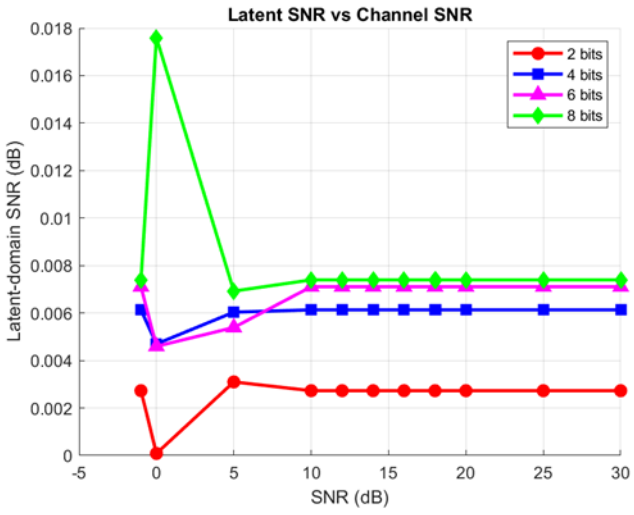

FIGURE 4.3: Latent SNR vs Channel SNR.

The Latent SNR vs Channel SNR figure showcases the behavior of the SNR in the latent aspect. It reflects the effective semantic signal power after channel corruption and quantization. Across all levels the latent SNR values remain very small (<0.02 dB), reflecting the difficulty of directly translating channel SNR improvements into semantic-domain gains. However an important trend is observed:

- Higher bit values such as 6-8 yield better latent SNR values particularly at low channel SNR.

By contrast the lower values confirm that much of the semantic signal can be destroyed regardless of the channel quality. This metric therefore complements MSE

and CosSim by quantifying semantic degradation in terms of effective information power.

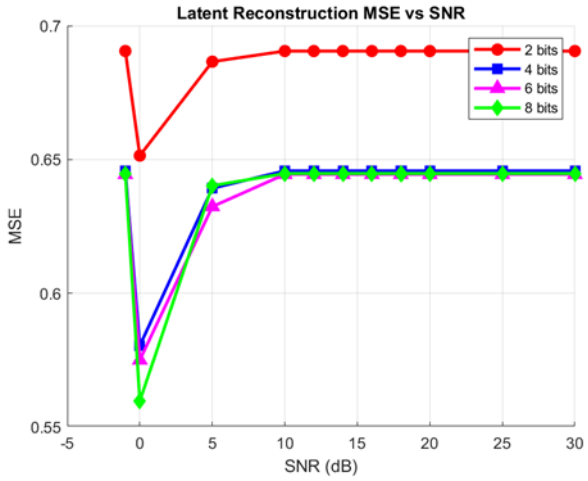

FIGURE 4.4: Latent Reconstruction MSE vs SNR.

The final figure illustrates the evolution of the Mean Squared Error (MSE) between transmitted and reconstructed latent vectors as a function of the channel SNR. The results demonstrate that the choice of quantization resolution plays a huge role on the effect of reconstruction fidelity. In particular for the 2 bit quantization case there is a consistent higher distortion compared to 4, 6, and 8 bit configurations. At very low SNR values (0–5 dB), some fluctuations are observed, but as the SNR increases, the MSE stabilizes. For higher quantization resolutions ($\geq$ 4 bits), the MSE converges around 0.64 which shows that channel noise becomes insignificant towards high SNR conditions. At the same time we can see that quantization effects are the primary source of distortion. This indication provides insight based on the semantic fidelity which is fundamentally constrained by quantization depth, rather than channel reliability, once a moderate SNR is reached.

Chapter 5

Conclusions and Future Work

Future extensions to further enhance the security and traceability of the semantic communication framework we can incorporate concepts inspired by blockchain technology. While blockchain (where we could take inspiration from the AES-CBC mode) is one of the most transformative technologies used today. In our implementation we simulate quantum key distribution (QKD) to secure the latent semantic vector using AES encryption in CBC mode. The very structure of AES relies on the previous natural conceptual link. Building on this intuition, a lightweight distributed ledger could be introduced to immutably log key exchanges, transmission events, and message metadata. Each hash of the encoded latent vector could be recorded alongside its timestamp and sender/receiver identity in combination with the technologies of quantum computing. Thereby semantic integrity verification can be applied. Such a hybrid approach with QKD-inspired encryption not only uses the blockchain but also extends them, providing accountability as it has been highlighted in recent research exploring quantum-secure blockchains.

Bibliography

Ayas, M. S., Ayas, S., and Djouadi, S. M. (2022). Projected gradient descent adversarial attack and its defense on a fault diagnosis system. In *Proc. 45th Int. Conf. Telecommunications and Signal Processing (TSP)*, pages 45–50.

Berger, T. (1971). *Rate Distortion Theory: A Mathematical Basis for Data Compression*. Prentice-Hall, Englewood Cliffs, NJ.

Bermparis, N. (2025). Semcom-qcrypto: End-to-end semantic communication over quantum-cryptographic secure channel. https://github.com/BERBARIANKING/SemCom-QCrypto. GitHub Repository. Accessed: 2025-10-29.

Burmester, M., Gritzalis, S., Katsikas, S., and Chrissikopoulos, V. (2011). *Syghroni Kryptografia: Theoria kai Efarmoges*. Klidarithmos, Athens. Original title: : .

Cover, T. M. and Thomas, J. A. (2006). *Elements of Information Theory*. Wiley, 2nd edition.

Gallager, R. G. (1968). *Information Theory and Reliable Communication*. Wiley.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples. In *Proc. ICLR*.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710.

Lu, Z., Hossain, E., Li, R., Lu, K., Chen, X., Zhao, Z., and Zhang, H. (2024). Semantics-empowered communications: A tutorial-cum-survey. *IEEE Communications Surveys & Tutorials*, 26(1):41–92.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *Proc. ICLR*.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA.

McInnes, L., Healy, J., and Melville, J. (2018). Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.

Meng, R., Gao, S., Fan, D., Gao, H., Wang, Y., et al. (2025). A survey of secure semantic communications. *arXiv preprint arXiv:2501.00842*.

Nguyen, T. and Tran, A. (2020). Input-aware dynamic backdoor attacks. *arXiv preprint arXiv:2012.10544*.

NIST (2001). Fips pub 197: Advanced encryption standard (aes). Technical report, National Institute of Standards and Technology.

NLTK Project (2024). Natural language toolkit (nltk) documentation.

Papineni, K., Roukos, S., Ward, T., and Zhu, W. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proc. ACL*.

Peng, X., Qin, Z., Huang, D., Tao, X., Lu, J., Liu, G., and Pan, C. (2022). A robust deep learning enabled semantic communication system for text. *arXiv preprint arXiv:2206.02596*.

Peng, X., Qin, Z., Tao, X., Lu, J., and Hanzo, L. (2024). A robust semantic text communication system. *IEEE Transactions on Wireless Communications*. Early access.

Prijatelj, D. S., Ventura, J., and Kalita, J. (2020). Neural networks for semantic textual similarity.

PyTorch Developers (2024). Pytorch: An open source machine learning framework.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656.

Shannon, C. E. (1959). Coding theorems for a discrete source with a fidelity criterion. *IRE National Convention Record*, 4:142–163.

Stacklok Labs (2023). Fgsm attack on llms. GitHub Repository.

TensorFlow Developers (2023). Adversarial example using fgsm. TensorFlow Tutorials.

van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605.

Weaver, W. (1949). Recent contributions to the mathematical theory of communication. In Shannon, C. E. and Weaver, W., editors, *The Mathematical Theory of Communication*. University of Illinois Press.

Wikipedia (2024a). Block cipher mode of operation. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

Wikipedia (2024b). Word error rate (wer). https://en.wikipedia.org/wiki/Word_error_rate.

Xia, R. (2024). *Semantic Communications with Knowledge Base Systems*. PhD thesis, University of Glasgow.

Xie, H., Qin, Z., Li, G. Y., and Juang, B.-H. (2022). A tutorial on semantic communications. *arXiv preprint arXiv:2212.08487*.

Xie, Y. (2021). Ece 587 lecture 18 - awgn channel models.

Zhang, T. et al. (2019). Bertscore: Evaluating text generation with bert. *OpenReview*.