INTRODUCTION

1. How NLP Relates to AI, ML, and DL

Before diving deep into NLP, ***it's helpful to understand how it fits within other tech fields.*** like Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL) as shown in Figure 1

- **Artificial Intelligence (AI)** is a broad field that aims to make machines think and act like humans. It covers areas like **machine learning**, **NLP**, **computer vision**, and **speech recognition**.
- **Machine Learning (ML)** is a branch of AI where computers learn patterns from data instead of being explicitly programmed. It includes:
  - **Supervised learning**: learning from labeled data
  - **Unsupervised learning**: finding hidden patterns in data
  - **Reinforcement learning**: learning through trial and error

- **Deep Learning (DL)** is a part of ML that uses **deep neural networks** (many layers stacked together) to handle complex problems like image recognition, language understanding, etc.

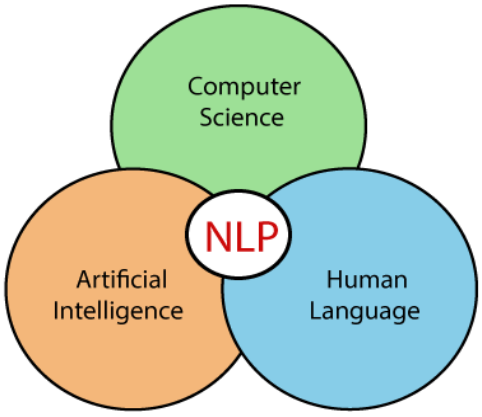


Figure 1: Relation of NLP with AI , ML and DL

Natural Language Processing (NLP) sits at the intersection of AI, ML, and DL. It enables machines to understand and generate human language.
Today's NLP heavily relies on **machine learning** and especially **deep learning** to handle tasks like translation, question-answering, and chatbots.
However, not all NLP requires ML — older methods like word counts or similarity measures can still be useful, even if they're not considered true "learning."

Related Fields

- **Computational Linguistics (CL)**: More focused on the **scientific study of language** using computational tools. It overlaps with NLP but has a more academic or research-driven goal.
- **Text Mining**: Focused on extracting meaningful patterns from **unstructured text data**, often used in business intelligence, and shares many tools with NLP.

NLP Vs NLU Vs NLG

- Natural Language Processing (NLP)
  NLP is a part of artificial intelligence that helps computers understand and work with human languages like English or Hindi. It was first introduced by Alan Turing around 1950. NLP goes through five steps: understanding words, sentence structure, meaning, how sentences connect, and the context. It takes input using sensors, processes it, and gives output. Some common uses of NLP are in smart assistants, language translation, and text analysis.

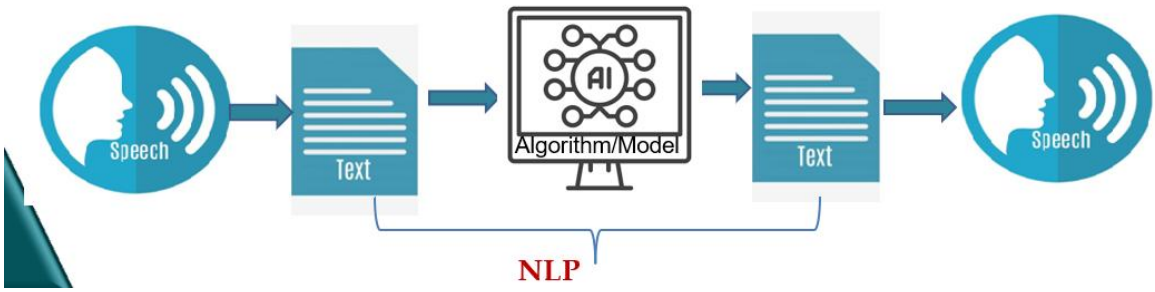
- Natural Language Understanding (NLU)
  NLU is a part of NLP that helps computers understand what we really mean when we speak or write. It started around 1866. NLU changes the unstructured text we give (like messages or speech) into useful and structured data. It works in three steps: rephrasing the input, translating it, and making conclusions. NLU is used in speech recognition, spam filtering, and finding emotions in text.

- Natural Language Generation (NLG)
  NLG helps computers create human-like sentences after understanding the input. It began around 1960. It works in three steps: understanding the input, planning what to say, and then forming the sentence. After processing, it gives the output using speakers or text. NLG is used in chatbots, voice assistants, and automatic report writing.

What Does NLP Do?

NLP takes **text written or spoken by humans** and:

- Finds useful information (like names or meanings)
- Classifies or labels text
- Translates from one language to another
- Predicts the words in Summarizes long paragraphs
- Generates new text or spoken words (like ChatGPT does! Or VA)



2. What is Natural Language Processing

- **Natural Language Processing (NLP)** is a part of **AI**

Definition: Natural Language Processing (NLP) is a field that combines computer science, artificial intelligence and language studies. It helps computers understand, process and create human language in a way that makes sense and is useful. With the growing amount of text data from social media, websites and other sources, NLP is becoming a key tool to gain insights and automate tasks like analysing text or translating languages.

- Human languages are the **means to communicate** by express thoughts, emotions, and ideas.
- As of **2025**, there are approximately **7,000 languages** spoken around the world.
- **Examples of Natural languages**
  - *Foreign Languages*
    English, French, German, Spanish, Italian, Russian, Mandarin Chinese, Japanese, Korean
  - *Indian Languages*
    Hindi, Bengali, Telugu, Marathi, Tamil, Gujarati, Kannada, Malayalam, Punjabi, Odia, Assamese, Urdu, Sanskrit, Maithili, Sindhi
- All humans speak with natural languages
  Example: To greet someone – Namaskar am, Aloha, Bonjour, hello, Ciao,.. etc.,

Why is it Called "Natural"?

The word **"natural"** means the language we **naturally speak and learn**, without formal rules.
Eg: English, Hindi, Spanish — all are **natural languages**.
In contrast, **formal languages** are **Created by humans** for specific tasks

- Very **strict and rule-based**
- Examples: Programming languages like **Python**, **C**, **Java**

Formal vs Natural Language

In *formal languages* (like "C"code), either something is correct, or it's not.

- If the code is wrong, you get an error.
- If it's right, it always does the same thing.

- In *natural language,* it's **not always clear**.
  - A sentence can be **half-correct**.
  - People **may disagree** on what's correct.
  - One sentence can have **many meanings**!

3. Why NLP?

- Computers can understand the **structured form of data** like spreadsheets and the tables in the database.
- But human languages, texts, and voices form an **unstructured category** of data, and it gets difficult for the computers to understand it, and there arises the need for Natural Language Processing.
- **Processing the Human languages** are interesting but challenging.
- Languages are classified based on historical and linguistic relationships.
- Human language is complex, ambiguous, disorganized, and diverse.
- There are more than 7,000 languages in the world, all of them with their own syntactic and semantic rules

Human languages are **ambiguous**, because their **interpretation** is often not unique.

- Both **structures** (*how sentences are formed*) and **semantics** (*what sentences mean*) can have ambiguities in human language.

4. NLP Components or Phases to analyse the ambiguities:

Natural Language Processing (NLP) helps computers to understand, analyse and interact with human language. It involves a series of phases or analysis as shown in figure.2 .in each phase it processes language and help to understand the structure and meaning of human language.

Figure 2: NLP Components or Phases of analysing the ambiguities

1. *Lexical Ambiguity*

Definition: Occurs when a single word has more than one meaning, and the intended meaning is unclear from the context.

A single word may `have multiple meanings (polysemy) or may refer to different entities (homonymy).

Challenge: Disambiguating word senses using Word Sense Disambiguation (WSD) techniques.

Cause:

- **Polysemy: One word with related meanings**
  → e.g., *"paper"* (a material vs. a newspaper vs. an academic article)
- **Homonymy: One word form with unrelated meanings**
  → e.g., *"bank"* (a financial institution vs. riverbank)

Example:

- ***"He went to the bank."*** → Is it a financial bank or a riverbank?

I saw a bat.

Example: *"bat"* → {animal, sports equipment}

2. *Syntactic Ambiguity*

Definition*: Occurs when a sentence can be parsed in multiple ways due to its structure or grammar. The sentence allows more than one syntactic interpretation.*

Challenge: *Building robust parsers (e.g., dependency or constituency parsers) to correctly interpret sentence structure.*

Cause*:*

- ***Multiple grammatical structures*** *possible for the same sequence of words*
- ***Ambiguity in phrase attachment*** *(e.g., prepositional phrases, modifiers)*
- ***Lack of punctuation or prosody*** *(especially in written language)*

Example*:*

- *"I saw the man with the telescope."*
  *→ Did **I** use the telescope, or did the **man** have it?*
- *"She watched the dog in the garden."*
  *→ Was the **dog in the garden**, or was **she** in the garden?*

" He Saw a girl with a telescope "

- *Is it the boy, who's using a telescope to see a girl (from somewhere far), or*
- *The girl, who has a telescope and is seen by the boy?*

There seem to be at least two interpretations of this sentence

3. *Semantic Ambiguity*

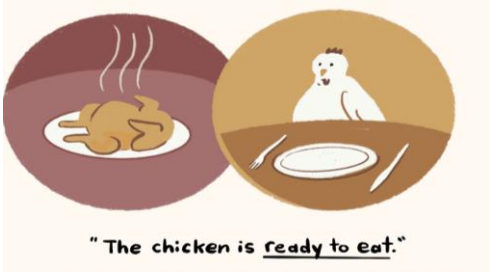
Definition: Occurs when a sentence has multiple meanings even though its grammatical structure is clear. The ambiguity arises from the meanings of the words or how they're interpreted together.

Challenge: Modeling semantic roles (who does what to whom) and predicate-argument structure accurately.

- **Cause:** Lexical ambiguity (a word having multiple meanings)
- Unclear role assignments between subject, verb, and object
- Multiple plausible interpretations of relationships between sentence elements

Example:

- "The chicken is ready to eat."
  → Is the chicken going to eat something, or is it ready to be eaten?
- "Visiting relatives can be annoying."
  → Is it annoying to visit relatives, or are the relatives who are visiting annoying?



→ *Who's eating whom?*

4. **Discourse Ambiguity**
   Definition: Occurs when there's confusion in understanding references or relationships between parts of a text or conversation, often across multiple sentences.
   Challenge: Maintaining **cohesion** and **reference clarity** across sentence boundaries.
   Cause: Vague referents, unclear transitions, or missing context between sentences.
   Example:
   "John told Tom he had won the prize."
   → **Who won** the prize: **John** or **Tom**?
   → Ambiguity arises due to unclear **referent resolution** in discourse.

5. ***Pragmatic Ambiguity***
   Definition: Ambiguity that arises when meaning depends on **context**, **speaker intention**, or **real-world knowledge**, beyond literal interpretation.
   Challenge: Requires **common-sense reasoning**, **tone detection**, and **contextual interpretation**.
   Cause: Same sentence can have different meanings in different **situations** or **contexts**.
   Example:
   "Can you pass the salt?"
   → Literally a question about ability, but **intended as a request** (based on pragmatic cues).
   "She put the book on the table and sat on it."
   → What is *"it"*? (Book or table?) → Needs **pragmatic context** to resolve.

6. ***Morphological ambiguity***
   - Definition: Morphological ambiguity happens when a word can be broken down into smaller parts (like prefixes, roots, or suffixes) in more than one way, leading to different meanings or grammatical roles.

   Example: "Unlockable"
   1. **un + lockable** → *not able to be locked*
      → *"The door is unlockable, so it can't be secured."*
   2. **unlock + able** → *able to be unlocked*
      → *"The safe is unlockable with the right code."*

5. Challenges in NLP:

- Handling single word having multiple meanings **by Word Sense Disambiguation (WSD)** techniques
- *correctly* ***interpret sentence structure*** *on Parsers of* ***grammatical and phrase attachment***
- ***Word order*** *from one language to another*
- Modelling **semantic roles** (who does what to whom) and predicate-argument structure
- **cohesion** and **reference clarity** across sentence boundaries *on* unclear transitions of text
- Requires **common-sense reasoning**, **tone detection**, and **contextual interpretation**.
- Developing **code-mixed language models** and **language identification** systems in **Multilingualism**
- Handling **spelling errors**, **slang**, **emojis**, and **non-standard grammar** on **Noisy and Unstructured Text**
- Optimizing **inference speed** without compromising accuracy in real time processing
- Building **cross-lingual embeddings**, **transfer learning** on **Low-Resource Languages**

6. Applications of NLP

1. Chatbots or QnAs system

Chatbots are computer programs that can talk to people. They are used on websites to help answer questions or give support. The first chatbot was called **ELIZA**, created in 1966. Today's chatbots use AI to talk in a human-like way and help with tasks.

2. Text Classification

Text classification means sorting text into different categories. For example, marking emails as spam or not spam. It helps computers understand large amounts of text quickly and correctly. With machine learning and deep learning, this process is more accurate.

3. Sentiment Analysis

Sentiment analysis finds out if people's opinions are **positive**, **negative**, or **neutral**. It is used for product reviews, movie feedback, and social media posts. One common method is the **Bag of Words** technique, where important words are used to judge sentiment.

4. Machine Translation

Machine translation changes text from one language to another. Google Translate is a good example. Today, with neural networks, translations are faster and better. You can even point your phone at a sign in another language and see the translation instantly.

5. Virtual Assistants

Virtual assistants like **Alexa**, **Siri**, or **Google Assistant** help us with daily tasks. They listen to our voice, understand it, and respond like a human. They can set alarms, make lists, and even talk to us. They use NLP to understand and respond correctly.

6. Speech Recognition

Speech recognition turns spoken words into text. It is used in voice assistants, typing by speaking, or transcribing audio. NLP helps understand what is being said, even if people speak differently.

7. Text Summarization

Text summarization takes long articles or documents and makes short summaries. It helps people read important information faster. It is useful for news, research papers, and legal documents.

8. Named Entity Recognition (NER)

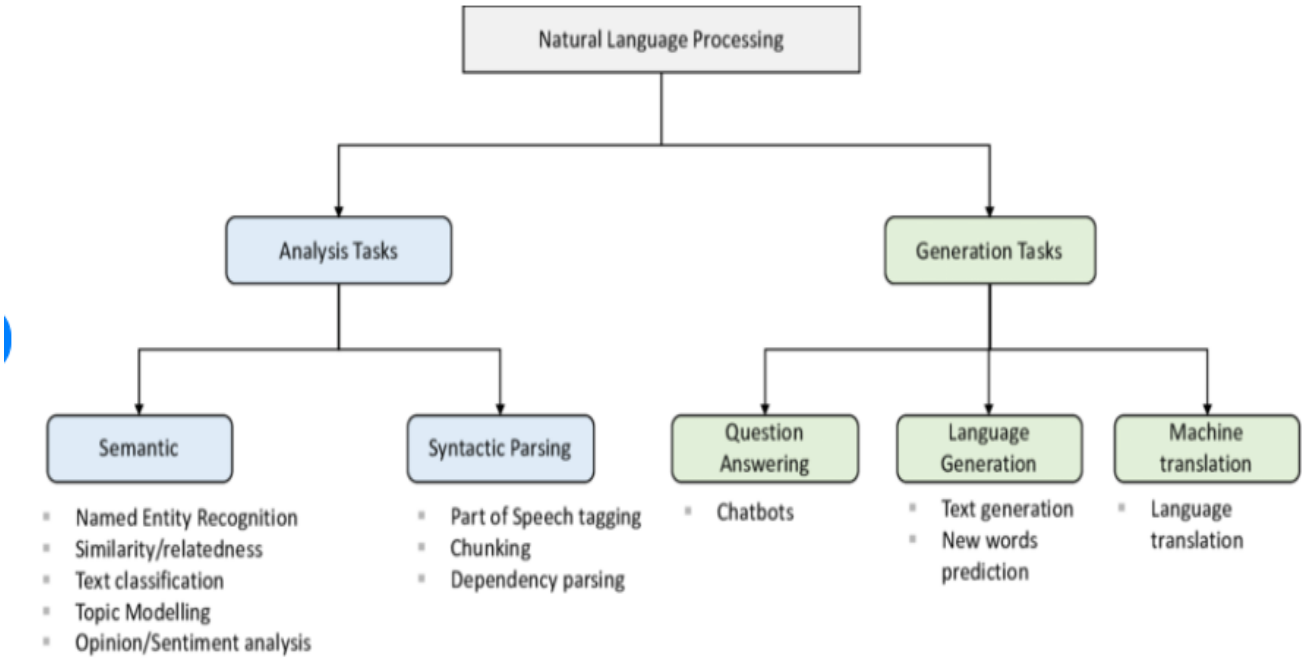
NER finds names of people, places, organizations, etc., in text. For example, in "Sachin plays for India," NER will detect *"Sachin"* as a person and *"India"* as a location. This helps search engines and chatbots understand key information.

10. Language Modeling

Language modeling is about predicting or generating text. It helps in typing suggestions, auto-complete, and even writing poems or stories. It is used in AI tools like ChatGPT or smart keyboards.

7. NLP Tasks

Modern NLP applications often integrate multiple components, each dedicated to solving a specific NLP task. These foundational tasks work together to power applications ranging from chatbots to search engines. Below are some of the most significant and commonly used NLP tasks.



a) Text Classification - Analysis

Text classification is the task of assigning predefined categories to pieces of text. It is one of the most basic yet widely applied NLP tasks. You may not recognize the term "text classification," but you encounter it regularly. For instance, **spam filtering** is a classic example where emails are classified into categories like "spam" and "not spam." This helps services like Gmail minimize spam in your inbox.

Another well-known use case is **sentiment analysis**, which involves identifying subjective information such as emotions, opinions, or feelings expressed in text. This type of classification is frequently used in product reviews, social media monitoring, and customer feedback systems.

b) Part-of-Speech (POS) Tagging

POS tagging is the process of labeling each word in a sentence with its corresponding part of speech—such as noun, verb, adjective, or preposition. A **part of speech** represents grammatical categories, and understanding them is essential for syntactic and semantic analysis.

For example, in the sentence **"I saw a girl with a telescope"**, each word is tagged as shown in **Figure 3**

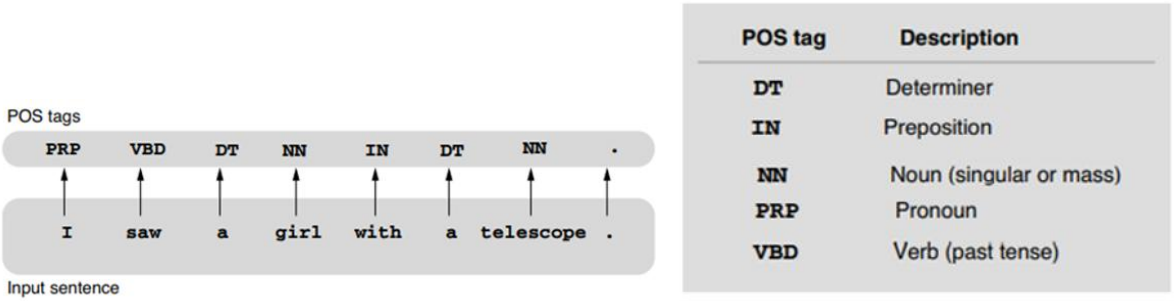


Fig 3: Part-of-Speech (POS) Tagging

The tags used—like **PRP** (pronoun), **VBD** (past tense verb), and **NN** (noun)—are part of the **Penn Treebank POS tagset**, which is widely used for training and evaluating POS taggers.

Traditionally, algorithms like Hidden Markov Models (HMMs) and Conditional Random Fields (CRFs) were used for POS tagging. Recently, **recurrent neural networks (RNNs)** have become popular due to their ability to learn from context and deliver higher accuracy. POS tagging results are often used as input for downstream tasks such as parsing and machine translation.

C) Parsing

Parsing involves analyzing the grammatical structure of a sentence. There are two main types: **constituency parsing** and **dependency parsing**.

i) Constituency Parsing

Constituency parsing uses **context-free grammars (CFGs)** to represent how smaller units of a sentence combine into larger structures like phrases and clauses. CFGs are defined using **production rules** such as:

```
S -> NP VP
NP -> DT NN | PRN | NP PP
VP -> VBD NP | VBD PN PP
PP -> IN NP
```

```
T -> a
IN -> with
NN -> girl | telescope
PRN -> I
VBD -> saw
```

Figure 4 & 5 Example for Parsing

These rules explain how parts of speech and phrases combine to form valid sentences. For example, using the rule **"NP → DT NN"**, the phrase **"a girl"** can be represented as a noun phrase (NP), as shown in **Figure 5**

By applying multiple production rules in reverse, a complete sentence can be parsed into a **parse tree**. In the sentence **"I saw a girl with a telescope,"** the parse tree can be built in multiple valid ways, shown in **Figures 1.6** and **1.7**.

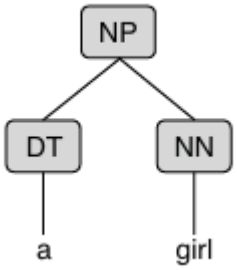


Fig6: Sub Tree of "A girl "

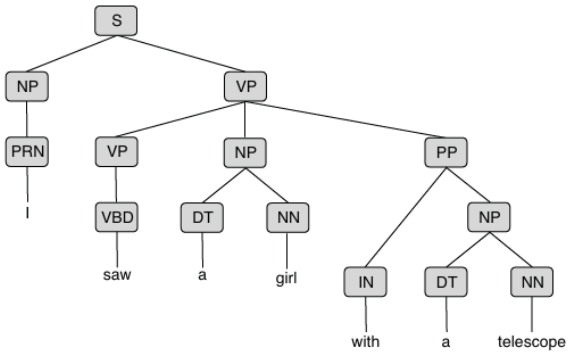


Fig7: parse tree

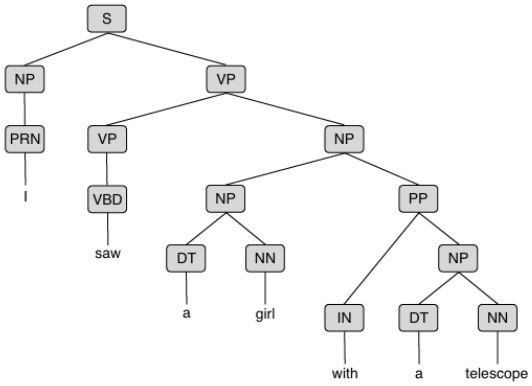


Figure 7: modified parse tree

These trees reveal **syntactic ambiguity**: in Figure 1.6, the phrase **"with a telescope"** modifies the verb "saw" (indicating the observer used a telescope), whereas in Figure 7, it modifies the noun "girl" (indicating the girl has a telescope). This highlights the limitations of parsing in resolving sentence meaning.

ii) Dependency Parsing

Dependency parsing takes a different approach by focusing on words and the binary relationships between them. It does not form phrase structures but instead builds **dependency trees**, where each word depends on another with a labeled and directional relation.

In **Figure 8**, the sentence **"I saw a girl with a telescope"** is parsed using dependency grammar. Each arrow shows the dependency between words.

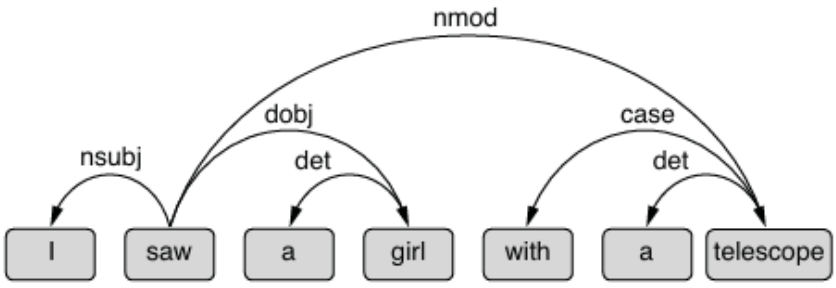


Figure 1.8 Dependency parse for "I saw a girl with a telescope."

For example, "a" is a **determiner (det)** of "girl," and "girl" is the **direct object (dobj)** of "saw." This method is especially useful in cases where word order varies without changing meaning. For instance, **"I carefully painted the house"** and **"I painted the house carefully"** result in the same dependency structure. Dependency parsing, therefore, captures fundamental word relationships and is crucial for deeper semantic analysis.

Ongoing work in this area includes the development of **Universal Dependencies**, a formal, language-independent dependency grammar applicable across multiple languages, much like the **universal POS tagset**.

d) Text Generation

Text generation, also known as **natural language generation (NLG)**, refers to the creation of natural language text from structured or unstructured input. This task comes in several forms.

i) Text-to-Text Generation

Many NLP applications such as **machine translation**, **summarization**, **text simplification**, and **grammatical error correction** fall under this category. These systems take one piece of text as input and produce another as output, often in a different form or language.

ii) Data-to-Text Generation

In these tasks, the input is structured data, not natural language. Examples include **dialog systems** that generate responses based on conversation state or **news generation systems** that turn sports or weather data into readable articles. Another example is **image captioning**, where models generate textual descriptions of images.

iii) Unconditional Text Generation

In unconditional generation, models produce random yet coherent text without a specific input. These models are trained to learn language patterns and generate text like stories, code, or even Shakespearean plays. A notable example is **Andrej Karpathy's** experiment, where he trained an **RNN** on Shakespeare's works. The generated output mimicked Shakespeare's style convincingly, as seen in the following excerpt:

Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
…

Traditionally, text generation was performed using handcrafted templates and grammar rules—essentially the reverse of parsing. Today, **neural network models** are the dominant approach across all types of text generation. This includes **sequence-to-sequence models** for text-to-text generation, **encoder-decoder models** for data-to-text generation, and **neural language models** and **GANs** for unconditional generation.

8.Building or development of NLP Application

The development of NLP applications is a highly iterative and cyclical process that involves several interdependent phases. While many books and tutorials focus on the model training phase, real-world NLP applications require attention across all stages—from data collection to monitoring. Developers, engineers, researchers, and other stakeholders often move back and forth among these phases based on experimentation and feedback, making it crucial to understand the overall structure before beginning development (Figure 9).

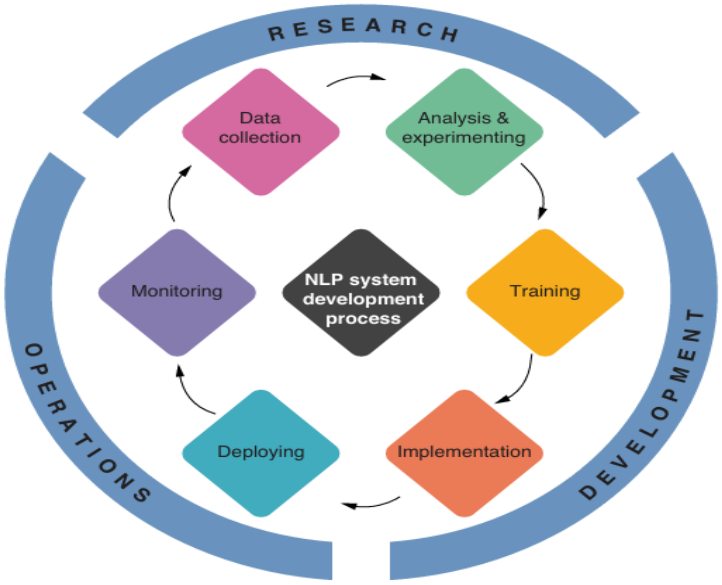


Figure 9: Three phases of Developing NLP Application

a) Data Collection

The first phase is *Data Collection*, which is foundational for any machine learning-based NLP system. At this stage, developers determine how to frame the problem and decide what kind of data they need. Data can be gathered in several ways, including manual annotation by experts, crowdsourcing via platforms like Amazon Mechanical Turk, or automatically through logs and user interactions. Even if you initially choose not to use ML, collecting a small dataset for validation is considered a best practice.

b) Analysis and Experimenting

Next comes ***Analysis and Experimenting***, where developers dive into the collected data to identify patterns, label distributions, and potential correlations. This phase involves asking critical questions such as: Should we use machine learning at all? Can rules alone solve the problem? What type of NLP task are we dealing with—classification, generation, or sequence labeling? Various prototype models or rule-based systems are quickly developed and evaluated to narrow down the most promising approaches. This research phase is essential because it helps avoid wasted effort on building full systems that may not work well in practice.

C) Training phase

After narrowing down approaches, developers move on to the ***Training phase***. Here, the selected model is trained using larger datasets and more computational resources such as GPUs. Training modern NLP models, especially neural networks, can take a considerable amount of time—days or even weeks. It is important to start small, test different model sizes, and monitor performance before committing to large-scale training. Maintaining a reproducible training pipeline is critical, as hyperparameter tuning and retraining are common over time.

D) *Implementation phase*

Once a working model is in place, the ***Implementation phase*** begins. This involves making the system production-ready by applying software engineering practices such as writing unit and integration tests, code refactoring, peer reviews, and containerization (e.g., using Docker). These steps ensure that the application is reliable, maintainable, and easy to deploy in various environments.

E) Deploying phase

In the ***Deploying phase***, the trained model is integrated into a real-world application. Deployment can take many forms, including online services, scheduled batch jobs, or one-time offline tasks. For real-time systems, it's often recommended to deploy the model as a microservice for better modularity and scalability. Using continuous integration (CI) practices ensures that the system remains functional and consistent as changes are made.

F) **Monitoring Phase**

Finally, the ***Monitoring phase*** ensures that both the infrastructure and the model continue to function as intended. This includes monitoring server health (CPU, memory, latency) as well as machine learning-specific metrics like input data distributions and prediction outputs. If the incoming data looks significantly different from the training data—a problem known as out-of-domain input—model performance may degrade. In such cases, the development cycle may need to restart from data collection to adapt to the new domain.

This structured, end-to-end development approach ensures that NLP applications are built efficiently, perform well in real-world settings, and remain adaptable as requirements and data evolve.

9.Structure of NLP Applications

Modern machine learning–based NLP applications are increasingly following similar structural patterns. This convergence is mainly due to two reasons. First, most modern NLP applications now incorporate machine learning, and therefore should adhere to best practices established in the ML domain. Second, the rise of neural network–based methods has enabled many complex NLP tasks—such as text classification, machine translation, dialogue systems, and speech recognition—to be trained end-to-end. Tasks that once required large, intricate systems with numerous components and pipelines can now often be handled with under a thousand lines of Python code, assuming there is sufficient training data.

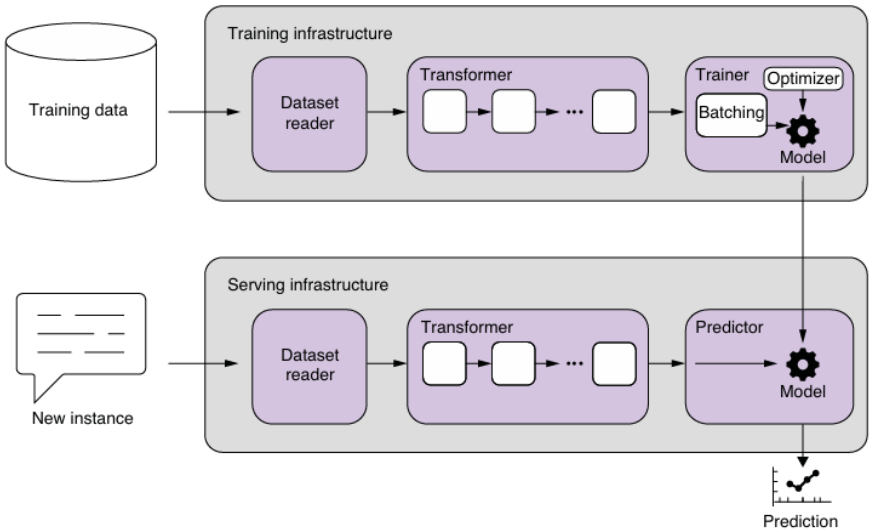


Figure 10 Structure of typical NLP applications

The Figure 10 illustrates the typical architecture of a modern NLP application, which is generally divided into two main infrastructures: **training** and **serving**.

The **training infrastructure** is typically used offline and is responsible for training the machine learning model. It begins by ingesting the training data, which is then converted into a format suitable for processing. This data is transformed and relevant features are extracted—a process that can vary significantly depending on the task. For neural network models, the data is usually batched and fed into the model, which is optimized by minimizing a loss function. The result of this process is a trained model that is serialized and stored for later use.

The **serving infrastructure**, on the other hand, handles real-time or batch predictions. When a new instance arrives, this infrastructure reads and processes it in a manner identical to the training phase. This is crucial because any mismatch between how the data is processed during training versus serving—known as **training-serving skew**—can lead to inaccurate predictions. Once the input instance is transformed correctly, it is passed through the pretrained model to generate outputs such as classification labels, tags, or translations. This end-to-end flow forms the backbone of many current NLP systems.

10. Introducing Sentiment Analysis

In some scenarios, like analyzing online surveys, users may answer open-ended questions without directly responding to structured prompts such as *"How do you like our product?"*. To interpret these subjective responses, **sentiment analysis** is used. This technique helps extract and classify emotions and opinions embedded in unstructured text.

Sentiment analysis is a method in natural language processing (NLP) that automatically identifies and categorizes subjective content in text, such as emotions, attitudes, and opinions. It is widely applied to user-generated content like reviews, surveys, and social media posts, where manual analysis is impractical due to volume.

In **machine learning**, sentiment analysis is treated as a **classification task**, where input text is sorted into predefined categories. The most common task is **polarity classification**, which determines whether the sentiment is **positive**, **negative**, or **neutral**. More detailed systems may use five categories (e.g., strongly positive to strongly negative), similar to star ratings on review sites.

Polarity classification is a form of **sentence classification**, much like **spam detection**, which sorts messages into **spam** or **not spam**. When there are only two categories, it's called **binary classification**. If there are more than two, like five-star ratings, it's a **multiclass classification** problem.

On the other hand, **regression** is used when the goal is to predict continuous values—such as estimating a house price based on its features or forecasting stock prices from online content. Despite its name, **logistic regression** is commonly used for classification. Since language elements are usually discrete, most NLP tasks, including sentiment analysis, rely more on classification than regression models.

NOTE: Despite its name, **logistic regression** is typically used for **classification tasks**.

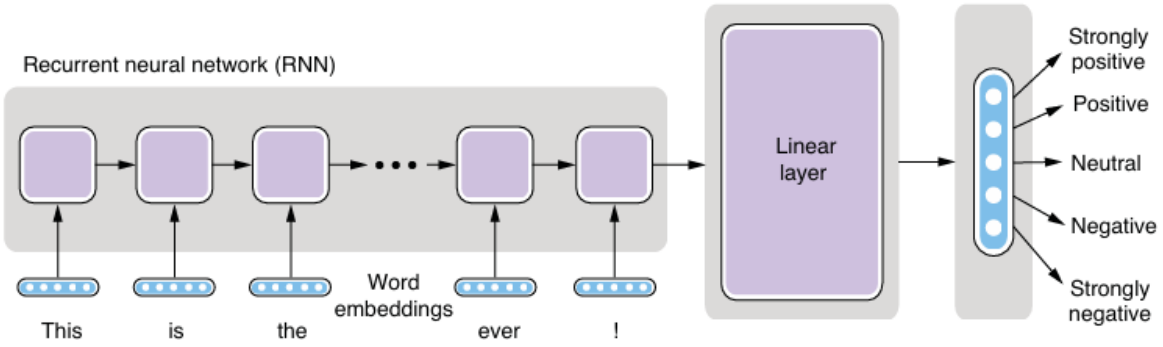


Fig 11: Sentiment analysis pipeline

Most modern NLP applications, including the **sentiment analyzer** we are going to build in this chapter (as illustrated in **figure 11**), are based on the **supervised machine learning paradigm**. In supervised learning, the algorithm is trained on data that includes **supervision signals**, which are the desired outputs associated with each input. The training process involves adjusting the model so it can predict these output labels as accurately as possible. For sentiment analysis, this means using datasets in which each sentence is labeled with its sentiment (e.g., positive or negative), allowing the model to learn how sentiment is expressed linguistically.

11. Working with Datasets:

Modern NLP applications often rely on **supervised machine learning**, where models are trained on data annotated with correct outputs, rather than being guided by manually written rules. Because of this, understanding how datasets are structured and used becomes essential in machine learning workflows.

What Is a Dataset?

A **dataset** is simply a structured collection of data. If you're familiar with relational databases, you can think of a dataset as being similar to a table, where each row is a **record** and each column is a **field**. In NLP, each record often represents a **linguistic unit**—a word, sentence, or document. A dataset of language texts is also known as a **corpus** (plural: **corpora**). As shown in **Figure 12.**

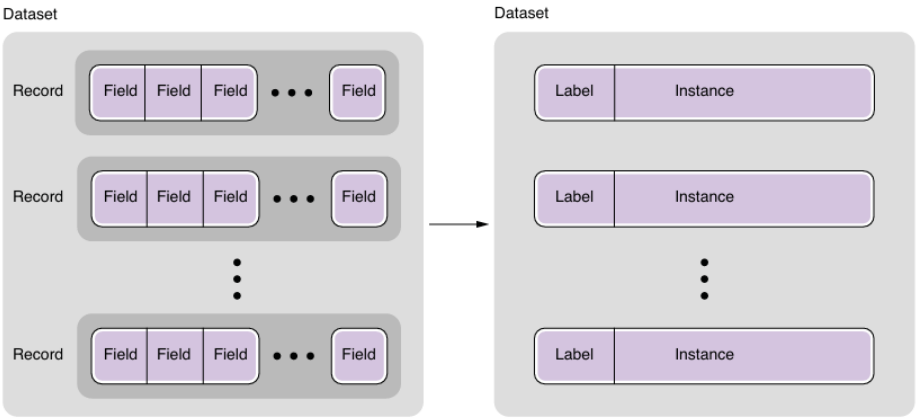


Figure 12 : Dataset for NLP Applications

For example, consider a dataset built for **spam filtering**. Each record might include a piece of text (e.g., an email sentence) and a label indicating whether it is spam or not. Both the text and the label are fields in the dataset.

Some NLP datasets are more complex, such as those annotated with **part-of-speech tags**, **parse trees**, or **semantic roles**. When a dataset includes syntactic annotations like parse trees, it's called a **treebank**. A well-known example is the **Penn Treebank (PTB)**, widely used in NLP research for tasks like POS tagging and parsing.

In machine learning, a **record** in a dataset often becomes an **instance**—the basic unit for which a prediction is made. In the spam filtering example, each email is an instance because a prediction (spam or not) is made for each email. However, depending on the task, an instance could also be a word—for example, in a model predicting whether each word is a noun.

Example Dataset: Stanford Sentiment Treebank (SST)

To build a **sentiment analyzer**, we'll use the **Stanford Sentiment Treebank (SST)**, a widely used dataset for sentiment analysis. SST is unique because it includes sentiment labels for not just sentences but also for every phrase and word within the sentence.

"The movie was actually neither that funny, nor super witty."

Although individual words like *funny* and *witty* are positive, the overall sentence conveys **negative** sentiment due to the structure "neither…nor." SST is thus a powerful resource for developing models that understand sentence structure. However, in this chapter, we focus only on sentence-level labels and ignore internal phrase annotations.

Train, Validation, and Test Sets

When building NLP models, it's important to evaluate them properly. A best practice is to split data into three sets: **train**, **validation**, and **test** (see **figure 13**).

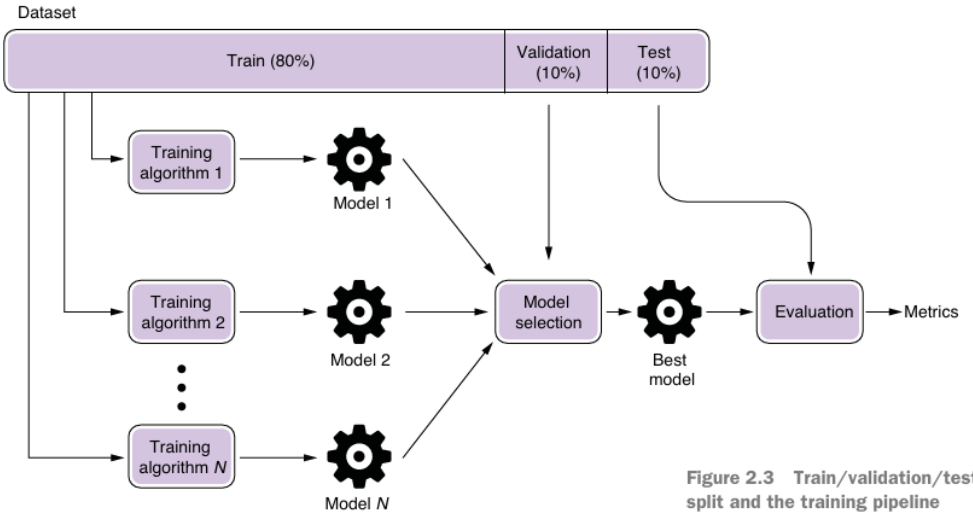


Figure 13 : Train/Validation/Test split

- The **train set** is used to train the model. It's usually the largest of the three and directly feeds the model's learning process.
- The **validation set** (or **dev set**) is used during training for **model selection**—that is, choosing among different algorithms or model versions. For instance, suppose you train two models, A and B. Instead of comparing them on the train set, which could lead to **overfitting**, you use the validation set to evaluate their generalizability.

Overfitting occurs when a model performs very well on training data but fails to generalize to new data. For example, a model that memorizes all training examples (like a giant lookup table) would score 100% on training data but fail to classify any new, slightly different input.

The **validation set** helps detect overfitting. It simulates real-world performance by being independent of the train set. It is also used for **hyperparameter tuning**, such as selecting the number of training epochs or the number of layers in a neural network.

- Finally, the **test set** is used to evaluate the final model. It contains completely unseen data to measure real-world performance. You shouldn't rely on the validation set for final evaluation because models can overfit to it as well, especially when many experiments are run using the same validation data.

For instance, if you run thousands of model variations and select the one with the best validation score, chances are it performs well on the validation set just by luck. This is why a separate **test set** is essential—it ensures a true measure of **generalization**.

In summary:

- Use the **train set** to train the model,
- Use the **validation set** to tune it and choose the best model,
- Use the **test set** to evaluate how it performs on completely new data.

Many NLP datasets, such as SST, come pre-split into these three sets. If your dataset is not split, you can manually divide it—commonly using an **80:10:10** ratio for train, validation, and test respectively.

Building our sentiment analyser with neural network architecture.

Architecture is just another word for the **structure** of neural networks—like the blueprint of a house. The first thing we need to do is figure out how to feed our input (in this case, sentences) into the network.

As discussed earlier, everything in NLP starts with **discrete symbols** (like words), which have no inherent numerical meaning. Neural networks, on the other hand, operate on **continuous**, numerical data—real-valued numbers. So, how do we "bridge" this gap between language and numbers?

The answer is through **word embeddings**, which we now explore in detail.

What Are Word Embeddings?

Word embeddings are one of the most fundamental breakthroughs in modern NLP. Technically, an **embedding** is a continuous **vector representation** of something that is originally discrete. A **word embedding** is just that—a vector representing a word.

If you're unfamiliar with vectors, think of them as **arrays of numbers**. A word embedding might be a vector with 100 or 300 values (or more), each one a float number. So instead of the word "cat" being represented as just the string "cat", it might be:

vec("cat") = [0.7, 0.5, 0.1, ..., 0.3]

Why is this important? In traditional symbolic NLP, we could assign **arbitrary indices** to words:

index("cat") = 1
index("dog") = 2
index("pizza") = 3

But this kind of assignment doesn't tell us anything about the **semantic similarity** between words. Just because "dog" has index 2 and "pizza" has index 3 doesn't mean they're similar.

That's where embeddings come in.

Let's imagine a **1-Dimensional embedding space** as in figure 14

Figure 14 Word embeddings in a 1-D space

This gives us some proximity information: maybe "dog" and "cat" are close. But **1D is limiting**. What if we added a second dimension?

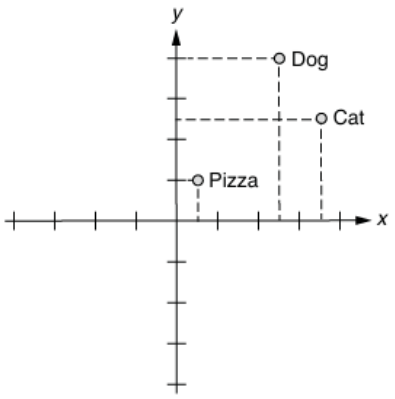


Figure 15 Word embeddings in a 2-D space

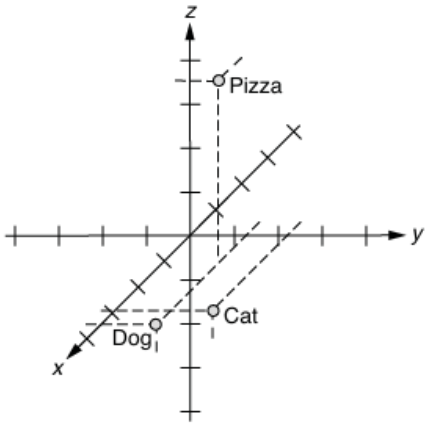


Figure 16 Word embeddings in a 3-D space

Now we can position "pizza" further away from "cat" and "dog" and give a more nuanced understanding. But we don't have to stop at two dimensions. In fact, we can go to **3 dimensions** (or more). As in figure 15 & 16

Here's how we might encode our three words in **3D**:

vec("cat") = [0.7, 0.5, 0.1]
vec("dog") = [0.8, 0.3, 0.1]
vec("pizza") = [0.1, 0.2, 0.8]

Each dimension can correspond to a concept like:

- **x-axis** = "animal-ness"
- **z-axis** = "food-ness"

So now we see that embeddings can represent **semantic meaning** in a multidimensional space. This is a huge improvement over arbitrary index.

One-Hot Encoding

Before embeddings, one simple method to "numerically" represent words was **one-hot encoding**. If we had 3 words in our vocabulary, it might look like this:

vec("cat") = [1, 0, 0]
vec("dog") = [0, 1, 0]
vec("pizza") = [0, 0, 1]

But the problem is, all words are equally distant from each other—no **semantic similarity** is captured. This is why **learned embeddings** (like GloVe or word2vec) are so powerful—they allow us to embed **meaning** into the model.

Neural Networks for NLP

Neural networks are a cornerstone of modern Natural Language Processing (NLP). Just as they've transformed fields like computer vision and game-playing AI, they now power many language-based applications such as sentiment analysis, machine translation, and question answering. In this section, we explore what neural networks are, why they are powerful, and how two specific components—Recurrent Neural Networks (RNNs) and Linear Layers—are used in NLP tasks such as sentence classification.

What Are Neural Networks?

At their core, **neural networks** (also known as **artificial neural networks**) are **mathematical models that map vectors to other vectors**. In other words, a neural network takes an input vector, performs a series of computations, and outputs another vector. What sets neural networks apart from ordinary functions is that they are **trainable**—they learn how to transform inputs to desired outputs by adjusting internal parameters known as **weights** through a process called **training**.

This ability to learn from data is what gives neural networks their power. During training, the network compares its output to the expected result using a **loss function**. Based on this comparison, it adjusts its weights to reduce the loss, gradually improving its predictions. Over time, the network "learns" the patterns in the data and becomes capable of making accurate predictions on new inputs.

Another key advantage of neural networks is their ability to model **nonlinear relationships**. A **linear function** changes its output proportionally to changes in its input (e.g., doubling the input doubles the output), but language doesn't behave this way. In natural language, changing a single word can dramatically alter the meaning of a sentence depending on context. Since neural networks are **nonlinear models**, they can capture these complex, non-proportional relationships. In fact, neural networks have been proven to be **universal approximators**—given enough data and capacity, they can learn to model any continuous function.

Recurrent Neural Networks (RNNs)

Neural networks are powerful models that can learn complex, nonlinear relationships from data, making them highly effective for NLP. Their strength lies in their trainability, flexibility, and ability to capture hidden patterns in sequential input like language. **Recurrent Neural Networks (Figure 17)** help model sentences by processing words sequentially and retaining contextual information, while **Linear Layers (Figure 18)** map these representations to output spaces such as sentiment classes. Together, they form the backbone of many state-of-the-art NLP systems used in real-world applications today.

A **Recurrent Neural Network (RNN)** is designed to process **sequential data** such as sentences, one element (e.g., word) at a time. It contains a **looping structure**, meaning it applies the same internal logic repeatedly across each step of the sequence. This structure allows the RNN to **retain memory** of previous elements, capturing context from earlier parts of the sequence.

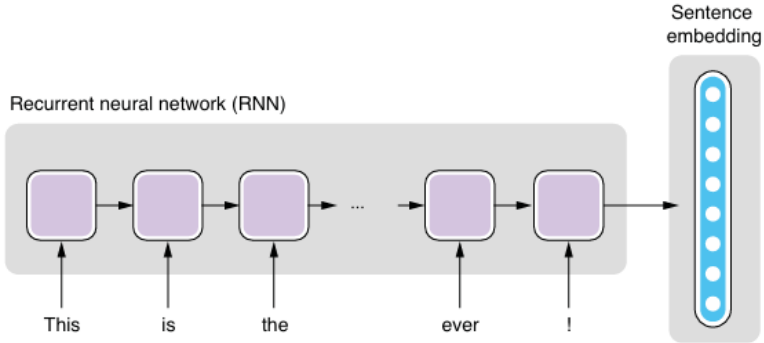


Figure 17: Recurrent Neural Network (RNN)

Imagine a sentence being processed word by word—an RNN updates its internal state after each word, building a **sentence embedding**: a fixed-length vector that summarizes the meaning of the entire sentence. This is especially useful for NLP tasks like sentiment analysis, where the sentiment of a sentence often depends on the overall structure and sequence of the words used.

Advanced versions of RNNs, such as **LSTM (Long Short-Term Memory)** and **GRU (Gated Recurrent Unit)**, are commonly used in practice because they handle long-range dependencies more effectively and avoid issues like vanishing gradients during training.

Linear Layers

A **Linear Layer** (also called a **fully connected layer**) is another essential component of neural networks. It performs a **linear transformation** of the input vector by multiplying it with a weight matrix and adding a bias. This operation is used to **compress** or **expand** the dimensionality of data as it flows through the network.

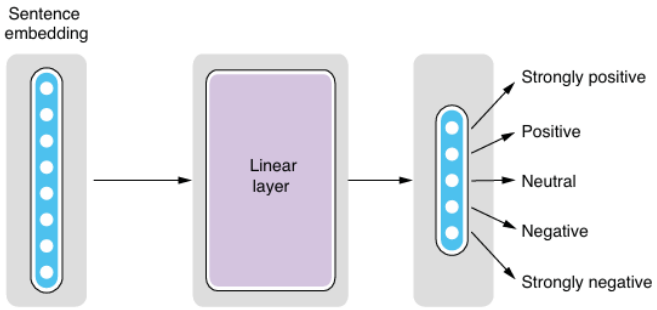


Figure 18: Linear Layer

For example, after an RNN produces a 64-dimensional vector representing a sentence, a linear layer can map this to a **5-dimensional output** corresponding to five sentiment categories: *strongly positive*, *positive*, *neutral*, *negative*, and *strongly negative*. The neural network learns how to perform this transformation during training, adjusting the weights of the linear layer to correctly associate features in the sentence embedding with output labels.

Although linear layers are limited in their expressiveness compared to nonlinear layers, they are **critical for output generation**, especially when converting high-dimensional internal representations into structured predictions (e.g., classification scores or probabilities).

Loss Functions and Optimization

Neural networks are trained using **supervised learning**, where the model learns to map inputs to correct outputs using a large number of labeled examples. So far, we've discussed how neural networks process input and produce output. But the real power of neural networks lies in their ability to **learn from mistakes** and improve over time. This learning process relies on two essential components: **loss functions**, which measure how wrong the model's predictions are, and **optimization**, which updates the model to reduce those mistakes.

Loss Functions: Quantifying Error

A **loss function** is a mathematical expression that calculates the difference between the predicted output and the true label. This difference, known as the **loss**, is what the model aims to minimize during training. The smaller the loss, the better the model is performing. Conversely, a large loss indicates poor predictions.

In **classification tasks**, such as predicting sentiment, the most common loss function is **cross-entropy loss**. It compares the predicted probability distribution over classes (e.g., positive, negative) to the true class distribution and penalizes incorrect or uncertain predictions.

For multi-class classification, the **cross-entropy loss** is given by:

$$\mathcal{L} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where:

C is the number of classes,

y_i is the true label (1 for correct class, 0 otherwise),

$\hat{y}_i$ is the predicted probability for class i.

For **binary classification**, the formula becomes:

$$\mathcal{L} = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$$

This loss is **zero** when the prediction is exactly correct and increases as the prediction becomes more wrong or uncertain.

Optimization: Learning from Loss

Once the loss is computed, the model must **adjust its internal parameters (weights)** to make better predictions in the future. This process is called **optimization**. It involves calculating how much each weight in the network contributed to the loss and then adjusting those weights in the direction that **reduces the loss**.

This is done using the method of **gradient descent**, where we compute the gradient (i.e., partial derivative) of the loss with respect to each parameter and take a small step in the opposite direction of the gradient:

$$w \leftarrow w - \eta \cdot \frac{\partial \mathcal{L}}{\partial w}$$

Here:

w is a weight,

η is the **learning rate**, which controls the size of the step,

$\frac{\partial \mathcal{L}}{\partial w}$ is the gradient of the loss with respect to the weight.

This update is repeated for each input-output pair in the training data. A complete pass over the entire training dataset is called an **epoch**. During training, the model usually goes through **multiple epochs**, continuously refining its weights.

There are several optimization algorithms built upon gradient descent. One popular variant is **Adam**, which adapts the learning rate individually for each parameter based on historical gradients, often resulting in faster convergence.

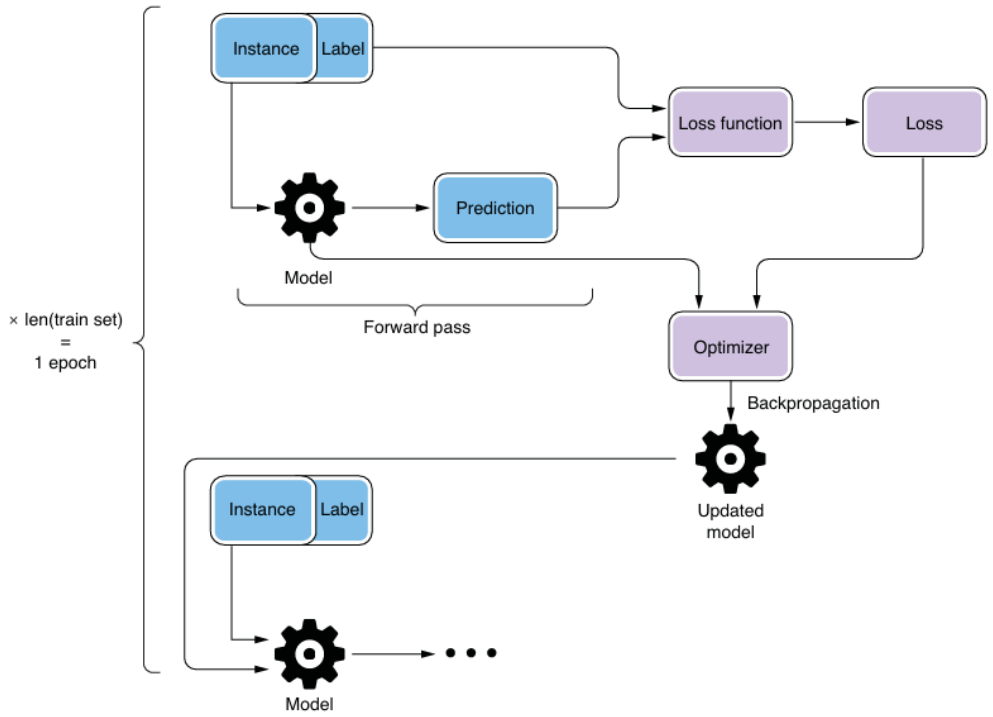


Figure 19 – Overall Training Procedure for Neural Networks

Figure 19 visually summarizes the entire training loop for neural networks:

1. An **input instance** and its **label** are provided to the model.
2. The model performs a **forward pass**, generating a prediction.
3. The **loss function** compares the prediction with the true label and calculates the loss.
4. Using **backpropagation**, the model computes the gradients of the loss with respect to each weight.
5. An **optimizer** (e.g., Adam or SGD) updates the weights to minimize the loss.
6. This process repeats for every example in the training set, completing one **epoch**.
7. Multiple epochs are performed to further refine the model.

$$\text{Loss} = \text{Prediction error per instance} \times \text{Number of instances} = 1 \text{ epoch}$$

This cycle of **forward pass → loss → backpropagation → optimization** is the core mechanism through which neural networks learn.

In summary, **loss functions** measure how far off the model's predictions are from the true answers, and **optimization** is the mechanism that adjusts the model's parameters to reduce this error. As illustrated in **Figure 2.9**, the process of computing predictions, measuring loss, and updating weights allows the network to gradually learn patterns from data. This iterative process is what enables modern neural networks to solve complex tasks such as language understanding and sentiment classification.

Training Your Own Classifier

Training a classifier using neural networks involves teaching a model to learn the relationship between inputs (such as sentences) and outputs (like sentiment labels). This is done through **supervised learning**, where the model is exposed to a dataset with known input-output pairs and learns to predict outputs for new, unseen inputs.

A crucial aspect of training is **batching**—the practice of grouping multiple data samples into a batch and processing them together during training. Rather than updating the model for every single data point, batching allows for more stable and efficient training. This is because data is often noisy—

individual examples might be mislabeled or ambiguous. By averaging the loss over a group of instances, batching reduces the influence of such noise on the training process.

In addition to improving stability, batching greatly enhances computational efficiency. Neural network training relies on large-scale matrix operations, and these are best executed on hardware like GPUs that can perform many computations in parallel. Feeding data in batches makes it possible to leverage the full power of such hardware, significantly speeding up training.

However, choosing the right **batch size** is important. If the batch is too small, it may not smooth out noise effectively. If it's too large, it might not fit in GPU memory or may slow down convergence. An optimal batch size balances both memory usage and training speed.

Evaluating Your Classifier

While training your model, it is critical to monitor its performance—not just in terms of training loss but also using **evaluation metrics** that better reflect real-world effectiveness. The **loss function** measures how far off the model's predictions are from the expected output, but a low loss doesn't always mean high-quality results on unseen data.

Commonly used evaluation metrics in classification tasks include:

- **Accuracy**: The proportion of correct predictions.
- **Precision**: The proportion of positive identifications that were actually correct.
- **Recall**: The proportion of actual positives that were correctly identified.
- **F1-Score**: The harmonic mean of precision and recall, offering a balanced measure.

These metrics help assess whether the model is learning meaningful patterns or simply memorizing the training data.

It's also important to compare metrics on the **training set** and a separate **validation set**. If a model performs well on training data but poorly on validation data, it is likely **overfitting**—learning noise or specific patterns from training examples that do not generalize to new inputs. Regular monitoring during training helps identify such issues early and adjust the model accordingly.

Deploying Your Application

Once the model is trained, the final step is **deployment**—making the model available for use in real-world applications. Deployment involves setting up the model so it can receive new inputs and return predictions, just like it did during training.

A key concept in deployment is ensuring that the **input pipeline used during prediction matches the one used during training**. Any deviation here could lead to **training-serving skew**, where the model behaves differently in deployment than it did during training, resulting in poor performance.

When deployed, the model receives raw inputs (such as text) and processes them through the same preprocessing and encoding steps as during training. The output is typically a set of **logits**—unnormalized scores representing confidence in each class. The class with the highest logit value is taken as the predicted label.

Once deployed, the model can be accessed via interfaces such as web applications or APIs. This allows users or other systems to interact with the model, providing inputs and receiving predictions in real-time. Monitoring the model post-deployment is also essential to ensure it continues performing well as it encounters new data in the wild.

we've explored the end-to-end lifecycle of building a neural network-based NLP classifier—from training with minibatches, monitoring with evaluation metrics, to finally deploying the model for real-world use. Even without diving into complex mathematical derivations or programming frameworks,

the high-level understanding of concepts like batching, optimization, overfitting, and deployment equips you with a solid foundation to appreciate the power and structure of modern NLP systems.

Steps for Building NLP Application:

Step 1. Data Collection: dataset relevant to the specific NLP task (e.g., text for SA,QA MT.

Step 2 .Data Cleaning: Remove noise, handle missing values, and correct errors in the data.

Step 3 Data Preprocessing- Tokenization; Stop Word; Removal; Stemming/Lemmatization

Step 4: Feature Engineering: word embeddings, TF-IDF, CBoW

Step 5: Model Selection: Choose an appropriate NLP model/ pre-trained models

Step 6: Training: training, validation, and test sets; adjust its parameters ; minimize the loss ; identify overfitting or underfitting.

Step 7: Evaluate the model's performance: Use test set evaluate model's generalization ability on unseen data.

Step 8. Generalization: Fine-tuning ; Data augmentation; Regularization techniques ; Ensemble methods, Transfer learning

Step 9. Deployment the model for real-world use; Monitor performance ; Retrain/refine the model; retrain the model with new data.

For Reference to Lab

---

spaCy sentiment classification pipeline using a CSV dataset with two columns:

Assumptions:

- Your CSV file is named sentiment_data.csv
- It has two columns:
  - "text" – contains the sentence
  - "label" – contains the sentiment ("POSITIVE" or "NEGATIVE")
  -

 # Step 1: Install and Import Dependencies

```
!pip install -U spacy pandas -q

import spacy
import random
import pandas as pd
import warnings
import pathlib
import shutil
from spacy.util import minibatch, compounding
from spacy.training.example import Example

warnings.filterwarnings("ignore")
```

 # Step 2: Load CSV Dataset with Two Columns

```
# Make sure the CSV file is in the same folder as your notebook
df = pd.read_csv("sentiment_data.csv")  # columns: text, label

# Preview
print(df.head())
```

 # Step 3: Convert DataFrame to spaCy Format

```
# Labels must be one-hot encoded for spaCy's textcat
train_data = []
for _, row in df.iterrows():
    label = row["label"].strip().upper()
    if label not in ("POSITIVE", "NEGATIVE"):
        continue  # skip bad labels
    cats = {"POSITIVE": 0.0, "NEGATIVE": 0.0}
    cats[label] = 1.0
    train_data.append((row["text"], {"cats": cats}))

print(f"Total training samples: {len(train_data)}")
```

 # Step 4: Create Blank spaCy Pipeline and Add TextClassifier

```
nlp = spacy.blank("en")
textcat = nlp.add_pipe("textcat", last=True)
textcat.add_label("POSITIVE")
textcat.add_label("NEGATIVE")
```

 # Step 5: Train the Model

```
n_epochs = 10
```

```
with nlp.select_pipes(enable="textcat"):
    optimizer = nlp.begin_training()
    for epoch in range(n_epochs):
        random.shuffle(train_data)
        losses = {}
        batches = minibatch(train_data, size=compounding(2.0, 16.0, 1.5))
        for batch in batches:
            texts, annotations = zip(*batch)
            examples = [Example.from_dict(nlp.make_doc(t), ann) for t, ann in zip(texts, annotations)]
            nlp.update(examples, drop=0.2, sgd=optimizer, losses=losses)
        print(f"Epoch {epoch+1}/{n_epochs} | Loss: {losses['textcat']:.4f}")
```

 # Step 6: Save the Trained Model

```
model_path = pathlib.Path("sentiment_model_csv_spacy")
if model_path.exists():
    shutil.rmtree(model_path)
nlp.to_disk(model_path)
print(f"\n Model saved to: {model_path.resolve()}")
```

 # Step 7: Load the Model and Make Predictions

```
nlp = spacy.load(model_path)
def predict_sentiment(text):
    doc = nlp(text)
    print(f"\nText: {text}")
    for label, score in doc.cats.items():
        print(f"{label}: {score:.4f}")
    pred = max(doc.cats, key=doc.cats.get)
    print(f"Predicted sentiment: {pred}")

# Try with examples
predict_sentiment("This product is amazing!")
predict_sentiment("Absolutely terrible and boring.")
```

NLTK

 # 1. Install & Import Libraries

```
!pip install pandas nltk -q

import pandas as pd
import nltk, random, string, pickle, os
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.classify import NaiveBayesClassifier
from nltk.classify.util import accuracy
from sklearn.model_selection import train_test_split
import tkinter as tk
from tkinter import scrolledtext

nltk.download('punkt')
nltk.download('stopwords')
```

```
# 2. Load & Clean Dataset
CSV_PATH = "amazon_reviews.csv"          # path to your CSV
TEXT_COL  = "text"                       # update if different
LABEL_COL = "label"                      # update if different

df = pd.read_csv(CSV_PATH)[[TEXT_COL, LABEL_COL]].dropna()

# lowercase labels and strip whitespace
df[LABEL_COL] = df[LABEL_COL].str.strip().str.upper()

# 3. Text Cleaning & Tokenization

stop_words = set(stopwords.words('english'))
punctuations = set(string.punctuation)

def clean_tokens(text):
    tokens = word_tokenize(text.lower())
    return [tok for tok in tokens if tok not in stop_words and tok not in punctuations]

def document_features(tokens):
    return {word: True for word in tokens}

# Convert dataframe rows to (features, label) tuples
featuresets = [
    (document_features(clean_tokens(row[TEXT_COL])), row[LABEL_COL])
    for _, row in df.iterrows()
]

# 4. Train/Test Split
random.shuffle(featuresets)
train_set, test_set = train_test_split(featuresets, test_size=0.2, random_state=42)

# 5. Train Naive Bayes Classifier
classifier = NaiveBayesClassifier.train(train_set)
print("\n Training complete.")

# 6. Evaluate
print(f"Test Accuracy: {accuracy(classifier, test_set)*100:.2f}%")
classifier.show_most_informative_features(10)



# 7. Save the Model
MODEL_FILE = "nltk_sentiment_model.pkl"
with open(MODEL_FILE, "wb") as f:
    pickle.dump(classifier, f)
print(f"\nModel saved to {MODEL_FILE}")

# 8. CLI Prediction Function
def predict_sentiment(text):
```

```
    tokens = clean_tokens(text)
    feats  = document_features(tokens)
    label  = classifier.classify(feats)
    prob   = classifier.prob_classify(feats)
    print(f"\nText: {text}")
    print("Predicted:", label)
    for lbl in prob.samples():
        print(f"{lbl}: {prob.prob(lbl):.4f}")

# Demo predictions
predict_sentiment("I absolutely loved this product, highly recommended!")
predict_sentiment("Terrible quality. Very disappointed.")
```

 # 9. Optional Tkinter GUI for Deployment

```
def launch_gui():
    def analyze():
        text = text_box.get("1.0", tk.END)
        tokens = clean_tokens(text)
        feats = document_features(tokens)
        label = classifier.classify(feats)
        prob  = classifier.prob_classify(feats)
        result.config(text=f"Prediction: {label} "
                      f"(Pos: {prob.prob('POSITIVE'):.2f}, "
                      f"Neg: {prob.prob('NEGATIVE'):.2f})")

    app = tk.Tk()
    app.title("NLTK Sentiment Analyzer")

    tk.Label(app, text="Enter Review:").pack()
    text_box = scrolledtext.ScrolledText(app, wrap=tk.WORD, width=60, height=10)
    text_box.pack(padx=10, pady=10)

    tk.Button(app, text="Analyze", command=analyze).pack(pady=5)
    result = tk.Label(app, text="Prediction: ")
    result.pack()
    app.mainloop()

# Uncomment to launch GUI
# launch_gui()
```

"""sentiment_fully_allennlp.py

End-to-end sentiment-analysis pipeline built **only with AllenNLP**:
• Dataset reader
• Model (embeddings + BiLSTM + linear classifier)
• Mini-batch training
• Archive saving
• Launch of AllenNLP REST server for deployment

Run once to train and save → then restart in --serve-only mode to host predictions.

pip install allennlp allennlp-models

```
import json, os, subprocess, torch, torch.nn.functional as F
from pathlib import Path
from allennlp.data import DatasetReader, Instance, Vocabulary
from allennlp.data.fields import TextField, LabelField
from allennlp.data.token_indexers import SingleIdTokenIndexer
from allennlp.data.tokenizers import SpacyTokenizer
from allennlp.data.data_loaders import MultiProcessDataLoader
from allennlp.modules.token_embedders import Embedding
from allennlp.modules.text_field_embedders import BasicTextFieldEmbedder
from allennlp.modules.seq2vec_encoders import PytorchSeq2VecWrapper
from allennlp.models import Model
from allennlp.nn.util import get_text_field_mask
from allennlp.training.metrics import CategoricalAccuracy, F1Measure
from allennlp.training.trainer import GradientDescentTrainer
from allennlp.training.optimizers import AdamOptimizer
from allennlp.predictors import SentenceClassifierPredictor
```

 # 1. Prepare tiny toy dataset

```
raw_data = [
    {"sentence": "I love this film", "label": "pos"},
    {"sentence": "What a great experience", "label": "pos"},
    {"sentence": "Horrible acting", "label": "neg"},
    {"sentence": "This movie was awful", "label": "neg"},
    {"sentence": "Absolutely fantastic", "label": "pos"},
    {"sentence": "Worst plot ever", "label": "neg"}]
DATA_JSON = "sentiment_data.json"
with open(DATA_JSON, "w") as fp:
    json.dump(raw_data, fp)
```

 # 2. AllenNLP DatasetReader

```
class SentimentReader(DatasetReader):
    def __init__(self) -> None:
        super().__init__(lazy=False)
        self.tokenizer = SpacyTokenizer()
        self.indexers = {"tokens": SingleIdTokenIndexer()}
    def text_to_instance(self, sentence: str, label: str | None = None) -> Instance:
        tokens = self.tokenizer.tokenize(sentence)
        fields = {"tokens": TextField(tokens, self.indexers)}
        if label is not None:
            fields["label"] = LabelField(label, label_namespace="labels")
        return Instance(fields)
    def _read(self, file_path: str):
        with open(file_path) as f:
            for item in json.load(f):
                yield self.text_to_instance(item["sentence"], item["label"])
reader = SentimentReader()
instances = list(reader.read(DATA_JSON))
```

```
# 3. Vocabulary
vocab = Vocabulary.from_instances(instances)

# 4. Model definition

class BiLstmSentiment(Model):
    def __init__(self, vocab: Vocabulary, embed_dim=64, hidden_dim=128):
        super().__init__(vocab)
        self.embedder = BasicTextFieldEmbedder(
            {"tokens": Embedding(vocab.get_vocab_size("tokens"), embed_dim)}
        )
        self.encoder = PytorchSeq2VecWrapper(
            torch.nn.LSTM(embed_dim, hidden_dim, batch_first=True, bidirectional=True)
        )
        self.fc = torch.nn.Linear(hidden_dim * 2, vocab.get_vocab_size("labels"))
        self.accuracy = CategoricalAccuracy()
        self.f1 = F1Measure(positive_label=vocab.get_token_index("pos", "labels"))

    def forward(self, tokens, label=None):
        mask = get_text_field_mask(tokens)
        x = self.embedder(tokens)
        vec = self.encoder(x, mask)
        logits = self.fc(vec)
        output = {"logits": logits}
        if label is not None:
            loss = F.cross_entropy(logits, label)
            self.accuracy(logits, label)
            self.f1(logits, label)
            output["loss"] = loss
        return output

    def get_metrics(self, reset: bool = False):
        return {
            "accuracy": self.accuracy.get_metric(reset),
            **self.f1.get_metric(reset)
        }

model = BiLstmSentiment(vocab)

# 5. DataLoader with batching

loader = MultiProcessDataLoader(
    reader, DATA_JSON, batch_size=2, shuffle=True, num_workers=0
)
loader.index_with(vocab)


# 6. Training

optimizer = AdamOptimizer(model.parameters(), lr=1e-3)

trainer = GradientDescentTrainer(
    model=model,
    data_loader=loader,
    validation_data_loader=None,
```

```
    optimizer=optimizer,
    num_epochs=10,
    cuda_device=-1,
)
print("Training…")
trainer.train()
```

 # 7. Save archive

```
MODEL_DIR = Path("sentiment_archive")
MODEL_DIR.mkdir(exist_ok=True)
archive_path = MODEL_DIR / "model.tar.gz"
model_path = MODEL_DIR / "weights.th"
vocab_dir = MODEL_DIR / "vocabulary"
torch.save(model.state_dict(), model_path)
vocab.save_to_files(vocab_dir)
print(f"Model saved under {MODEL_DIR}")
```

 # 8. Quick inference demo

```
predictor = SentenceClassifierPredictor(model, dataset_reader=reader)
print("Example prediction:",
      predictor.predict("I absolutely hated this film")["logits"])
```

 # 9. Deployment helper: start REST server

```
def launch_server():
    cmd = [
        "allennlp", "serve",
        "--archive-path", str(archive_path),
        "--predictor", "sentence-classifier",
        "--field-name", "sentence",
        "--port", "8000"
    ]
    print("Starting AllenNLP REST server at http://localhost:8000")
    subprocess.run(cmd)
```

```
# Uncomment to launch immediately after training:
# launch_server()
```

How to Use-Run the script once to train and save the archive:

```
python sentiment_fully_allennlp.py
```

Deploy-Uncomment launch_server() at bottom or run separately:

```
bash
allennlp serve \
  --archive-path sentiment_archive/model.tar.gz \
  --predictor sentence-classifier \
  --field-name sentence
```

Send a test request:

```
bash
curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"sentence": "What a fantastic masterpiece!"}' \
  http://localhost:8000/predict
```

The script demonstrates model creation, batching, training, archiving, and a turnkey REST interface—all done with AllenNLP only.

Optional: Deploy on Render, Heroku, or Hugging Face Spaces

if you'd like a you can do Render **or Hugging Face Spaces deployment template**, or **Docker support** for cloud deployment.