Fundamentals of NLP

AICL 434 | KU | 2025

Ashok Kumar Pant

CTO | Co-Founder @ Treeleaf.AI | Anydone

asokpant@gmail.com | https://www.linkedin.com/in/asokpant | https://github.com/ashokpant

Content

Chapter 1: Fundamentals of Natural Language Processing (NLP)

1.1. Introduction to NLP and its applications
1.2. Tokenization, Lemmatization, and Stemming
1.3. Part-of-Speech (POS) Tagging and Named Entity Recognition (NER)
1.4. Word Embeddings: Tf-idf, Word2Vec, GloVe
1.5. Sequence-to-Sequence Language Learning: RNNs
1.6. Applications of Encoder-Decoder Models in NLP

Natural Language Processing (NLP)

- NLP is a subfield of AI that enables computers to understand and process human language.
- It combines linguistics, computer science, and artificial intelligence.
  - *Computational Linguistics – Rule-based modeling of human language.*
  - *Machine Learning – Learning from data without explicit programming.*
  - *Deep Learning – Neural networks for advanced language tasks.*
- Used in applications like chatbots, translation, sentiment analysis, and more.

Colorless green ideas sleep furiously.

Evolution of NLP

Early NLP (1950s - 1970s): Rule-Based Systems

- **1950:** Alan Turing proposes the **Turing Test** to evaluate machine intelligence.
- **1957:** Noam Chomsky introduces **Transformational Grammar**, influencing computational linguistics.
- **1966:** Joseph Weizenbaum develops **ELIZA**, an early chatbot using rule-based responses.
- **1970s:** Development of **shallow parsing** and early **machine translation (MT)** systems.

Statistical NLP (1980s - 1990s): Probabilistic Models

- **1980s:** Introduction of **Hidden Markov Models (HMMs)** for speech recognition and **Part-of-Speech (POS) tagging**.
- **1990:** IBM develops the **Candide** statistical machine translation system.
- **1999:** Statistical models, such as **n-grams** and **log-linear models**, become popular for NLP tasks.

Evolution of NLP

Machine Learning-Based NLP (2000s - 2010s): Supervised Learning

- **2001:** Maximum Entropy Models improve NLP tasks like Named Entity Recognition (NER).
- **2003:** Latent Dirichlet Allocation (LDA) is introduced for **topic modeling**.
- **2008:** Support Vector Machines (SVMs) and **Conditional Random Fields (CRFs)** enhance text classification and sequence labeling.

Deep Learning Revolution (2010s - Present): Neural Networks and Transformers

- **2013:** Word2Vec by Google introduces **word embeddings**, improving contextual understanding.
- **2014:** Seq2Seq models enable high-quality machine translation.
- **2015:** Attention Mechanism enhances NLP tasks.
- **2017:** Google introduces **Transformer architecture** in "Attention is All You Need".
- **2018:** OpenAI releases **GPT (Generative Pre-trained Transformer)**, followed by BERT from Google.
- **2020s:** Models like **GPT-4, PaLM, T5, and LLaMA** revolutionize conversational AI, text generation, and RAG-based NLP.

Evolution of NLP

Future Trends in NLP

- **Multimodal NLP**: Integrating text with images, audio, and video.
- **Efficient NLP**: Lighter, faster models using **quantization and distillation**.
- **Explainable AI (XAI)**: Enhancing transparency in NLP decision-making.

Applications of NLP

- Text Processing & Understanding (Core NLP Tasks)
  - Tokenization, POS tagging, NER, sentiment analysis, summarization, topic modeling
- Conversational AI & Chatbots
  - Virtual assistants, customer support, AI-powered email responses, call center automation
- Machine Translation
  - Google Translate, real-time speech translation, cross-language search
- Speech Processing & Voice Assistants
  - Speech-to-text (ASR), text-to-speech (TTS), voice authentication
- Information Retrieval & Search Engines
  - Semantic search, auto-suggest, smart document retrieval
- Text Generation & Content Creation
  - AI writing assistants, code generation, creative writing
- Document Analysis & Processing
  - OCR, invoice/resume parsing, legal document analysis
- Healthcare & Biomedical NLP
  - Clinical NLP, AI mental health chatbots, drug discovery

Text Processing

Text processing is a fundamental task in natural language processing (NLP) and computer science, involving the manipulation, analysis, and transformation of textual data. It's a crucial step in various applications, including:

- **Search engines** Analyzing text to extract keywords and provide relevant search results.
- **Machine translation** Translating text from one language to another.
- **Sentiment analysis** Determining the sentiment (positive, negative, or neutral) expressed in text.
- **Information extraction** Identifying specific entities (e.g., names, dates, locations) from text.
- **Text summarization** Creating concise summaries of lengthy text documents.

Tokenization

Tokenization is the process of breaking down a text into individual units, or tokens, which can be words, phrases, or symbols. This step is crucial as it transforms unstructured text into a structured format that can be analyzed. Effective tokenization helps in identifying meaningful components of the text while discarding irrelevant characters or symbols.

Word Tokenization: Splitting text into words. For example, "Text preprocessing is essential" becomes ["Text", "preprocessing", "is", "essential"].

Subword Tokenization: Dividing words into smaller units. This is used in some deep learning models like BPE (Byte Pair Encoding). Example: "hello" might be tokenized as "he", "ll", "o".

Sentence Tokenization: Dividing text into sentences.

Stemming

Stemming is a technique used to reduce words to their base or root form. For instance, "running," "ran," and "runner" may all be reduced to "run." This process helps in consolidating different forms of a word into a single representation, which can enhance the performance of classification algorithms by reducing dimensionality. However, stemming may sometimes produce non-words or incorrect roots, which can affect analysis accuracy.

Popular Stemmer Algorithms:

- Porter Stemmer
  - Developed by **Martin Porter** in **1980**.
  - One of the earliest and most widely used stemming algorithms.
  - Uses a set of **heuristic rules** to iteratively remove common word endings (e.g., *running* → *run*, *happiness* → *happi*).
  - **Aggressive stemming** can sometimes produce non-intuitive stems (*e.g., "organization"* → *"organ"*) due to its rule-based approach.

- Snowball Stemmer (Porter2 Stemmer)
  - An **improved** version of the Porter Stemmer, also developed by **Martin Porter** in **2002**.
  - Supports multiple languages, including **English, Spanish, French, German, etc.**.
  - More **sophisticated and accurate** than the original Porter Stemmer.
  - Reduces over-stemming issues found in Porter Stemmer.

Stemmers

```
# Porter Stemmer

from nltk.stem import PorterStemmer
ps = PorterStemmer()
words = ["running", "flying", "happily", "organization",
"generalization", "relational", "troubling", "singing"]
for word in words:
   print(f"{word}->{ps.stem(word)}")

# running->run
# flying->fli
# happily->happili
# organization->organ
# generalization->gener
# relational->relat
# troubling->troubl
# singing->sing
```

```
# Snowball Stemmer

from nltk.stem import SnowballStemmer
ss = SnowballStemmer("english")
words = ["running", "flying", "happily", "organization",
"generalization", "relational", "troubling", "singing"]
for word in words:
   print(f"{word}->{ss.stem(word)}")

# running->run
# flying->fli
# happily->happili
# organization->organ
# generalization->general
# relational->relat
# troubling->troubl
# singing->sing
```

Lemmatization

Lemmatization also reduces words to their base form, but unlike stemming, it uses linguistic rules and ensures that the reduced form is a valid word (lemma).

For example, "better" would be lemmatized to "good.", Similarly, "Running" → "run". This technique tends to preserve the meaning of words better than stemming but requires more computational resources due to its reliance on vocabulary and morphological analysis. Lemmatization is particularly useful in applications where semantic accuracy is crucial.

WordNet Lemmatizer is a popular tool for this process

Wordnet Lemmatizer

```
import nltk
from nltk.stem import WordNetLemmatizer

nltk.download('wordnet')
wl = WordNetLemmatizer()
words = ["running", "flied", "goes", "organization", "generalizations", "relational", "troubling", "singing", ]
for word in words:
   print(f"{word}->{wl.lemmatize(word)}")

# running->running
# flied->flied
# goes->go
# organization->organization
# generalizations->generalization
# relational->relational
# troubling->troubling
# singing->singing
```

POS Tagging

- Part of Speech (POS) Tagging is a fundamental technique in Natural Language Processing (NLP) that involves assigning grammatical categories to each word in a sentence.
- POS Tagging labels each words with their appropriate grammatical categories, such as: Noun, Verb, Adjective, Adverb, Preposition, Pronoun, Conjunction, Interjection, etc.
- POS is the base for Named Entity Resolution, Sentiment Analysis, Question Answering, and Word Sense Disambiguation.

POS Tagging Approaches

Rule-Based Tagging

In this approach, manually created rules are used to tag words based on their linguistic characteristics.

- Predefined grammatical rules
- Manually constructed tag assignment criteria
- Typically less flexible and adaptable
- Requires extensive linguistic expertise

Stochastic-Based Tagging

This method uses probabilistic algorithms to determine the most likely part of speech for a word based on its context and sequence.

- Uses statistical models like Hidden Markov Models (HMM)
- Considers word sequence and context
- Calculates probability of tag sequences
- More adaptive and context-aware
- Can handle ambiguous word classifications

The stochastic approach is generally more advanced and widely used in modern NLP systems due to its ability to handle complex linguistic variations and context-dependent tagging.

POS Tags

1. CC coordinating conjunction
2. CD cardinal digit
3. DT determiner
4. EX existential there (like: “there is” ... think of it like “there exists”)
5. FW foreign word
6. IN preposition/subordinating conjunction
7. JJ adjective ‘big’
8. JJR adjective, comparative ‘bigger’
9. JJS adjective, superlative ‘biggest’
10. LS list marker 1)
11. MD modal could, will
12. NN noun, singular ‘desk’
13. NNS noun plural ‘desks’
14. NNP proper noun, singular ‘Harrison’
15. NNPS proper noun, plural ‘Americans’
16. PDT predeterminer ‘all the kids’
17. POS possessive ending parent’s
18. PRP personal pronoun I, he, she
19. PRP$ possessive pronoun my, his, hers
20. RB adverb very, silently
21. RBR adverb, comparative better
22. RBS adverb, superlative best
23. RP particle give up
24. TO to go ‘to’ the store
25. UH interjection
26. VB verb, base form take
27. VBD verb, past tense took
28. VBG verb, gerund/present participle taking
29. VBN verb, past participle taken
30. VBP verb, sing. present, non-3d take
31. VBZ verb, 3rd person sing. present takes
32. WDT wh-determiner which
33. WP wh-pronoun who, what
34. WP$ possessive wh-pronoun whose
35. WRB wh-adverb where, when

POS Tagger

```
import nltk

# Download required NLTK resources (run this once)
nltk.download('punkt')  # for tokenization
nltk.download('averaged_perceptron_tagger')  # for POS tagging

text = "I love NLP and I will learn NLP in 2 month. I am learning NLP"

sentences = nltk.tokenize.sent_tokenize(text)

for sentence in sentences:
    words = nltk.tokenize.word_tokenize(sentence)
    tags = nltk.pos_tag(words)
    print(tags)
# [('I', 'PRP'), ('love', 'VBP'), ('NLP', 'NNP'), ('and', 'CC'), ('I', 'PRP'), ('will', 'MD'),
# ('learn', 'VB'), ('NLP', 'NNP'), ('in', 'IN'), ('2', 'CD'), ('month', 'NN'), ('.', '.')]
# [('I', 'PRP'), ('am', 'VBP'), ('learning', 'VBG'), ('NLP', 'NNP')]
```

https://explosion.ai/blog/part-of-speech-pos-tagger-in-python

Named Entity Extraction(NER)

Named Entity Recognition is a natural language processing (NLP) technique used to automatically identify and classify named entities in text into predefined categories.
Named Entities: These are specific types of words or phrases that represent:

- People's names
- Organizations
- Locations
- Dates and times
- Monetary values
- Percentages
- Product names

Purpose of NER:

- Extract structured information from unstructured text
- Help in information retrieval
- Support text analysis and summarization
- Aid in building knowledge graphs
- Assist in various applications like search engines, chatbots, and document processing

Example:
In the sentence "Apple Inc. was founded by Steve Jobs in Cupertino, California in 1976", a NER system would identify:
Organization: Apple Inc.
Person: Steve Jobs
Location: Cupertino, California
Date: 1976

NER with NLTK

```
import nltk
from nltk.chunk import ne_chunk
from nltk.tag import pos_tag
from nltk.tokenize import word_tokenize

# Download necessary NLTK data
nltk.download("punkt")
nltk.download("averaged_perceptron_tagger")
nltk.download("maxent_ne_chunker_tab")
nltk.download("words")
text = "Elon Musk is the CEO of Tesla and SpaceX, and he was born in South Africa."

# Tokenize and Part-of-Speech (POS) tagging
words = word_tokenize(text)
pos_tags = pos_tag(words)

tree = ne_chunk(pos_tags)

for subtree in tree:
    if hasattr(subtree, "label"):
        entity_name = " ".join([token for token, pos in subtree.leaves()])
        entity_type = subtree.label()
        print(f"{entity_name} - {entity_type}")

# Elon - PERSON
# Musk - ORGANIZATION
# CEO of Tesla - ORGANIZATION
# SpaceX - ORGANIZATION
# South Africa - GPE
```

NER with Spacy

```
!pip install -q spacy
!python -m spacy download en_core_web_sm --quiet

import spacy

nlp = spacy.load("en_core_web_sm")

text = "Elon Musk is the CEO of Tesla and SpaceX, and he was born in South Africa."
doc = nlp(text)

print("Entities, their labels, and explanations:")
for ent in doc.ents:
  print(f"{ent.text} - {ent.label_} ({spacy.explain(ent.label_)})")
#
# Entities, their labels, and explanations:
# Elon Musk - PERSON (People, including fictional)
# Tesla - ORG (Companies, agencies, institutions, etc.)
# SpaceX - PERSON (People, including fictional)
# South Africa - GPE (Countries, cities, states)
```

Word Embeddings

One Hot Encoding - FYI

One-hot encoding is a simple and intuitive method to represent categorical variables (words or tokens) as binary vectors. Each unique word is represented as a vector where only one element is "1" and all others are "0."

	I	love	NLP	is	future
I love NLP	1	1	1	0	0
NLP is future	0	0	1	1	1

For the words ["cat", "dog", "fish"], one-hot encoding will generate

cat → [1, 0, 0] dog → [0, 1, 0] fish → [0, 0, 1]

TF-IDF Vectorization

Term Frequency-Inverse Document Frequency (TF-IDF) is an enhancement of the Countvectorizer. It assigns weights to words based on their importance in a document relative to the entire corpus. The idea is to down-weight common words and emphasize rare, informative words.

Term Frequency:

Term Frequency quantifies how often a word appears in a document relative to the total number of words in that document. It helps normalize the frequency of words, making it easier to compare their importance across documents of different lengths.

$$TF(t,d)=Count(t,d)/\ Total\ terms\ in\ d$$

Where

- Count(t,d) is the number of times term t appears in document d
- Total terms in d is the total number of words in document d

TF-IDF Vectorization

Consider the sentence: "The cat sat on the mat."

- Total words = 7
- Count of "the" = 2

Calculating TF for "the": $TF(\text{ the },d)=2/7\approx0.29$

If we had a longer sentence like: "The cat sat on the mat and looked around the house."

- Total words = 12
- Count of "the" = 3

Calculating TF for "the": $TF(\text{ the },d)=3/12=0.25$

This shows how TF helps adjust the importance of a term based on its frequency relative to the document length.

TF-IDF Vectorization

Inverse Document Frequency (IDF)

IDF measures how important a term is within the entire corpus. It reduces the weight of common words that appear in many documents and increases the weight of rare words that are more informative.

$$IDF(t)=\log(N/n)$$

Where, N is the total number of documents, n is the number of documents containing the term t

TF-IDF Vectorization

Suppose we have a corpus with 5 documents:

1. "The cat sat."
2. "Dogs are great."
3. "The cat and dog."
4. "The dog barked."
5. "The cat is playful."

Now, let's calculate IDF for the word "cat":

- Total documents N=5
- Documents containing "cat" n=3

Calculating IDF for "cat": $IDF(\text{ cat }) = \log(5/3) \approx 0.2218$

For a more common word like "the":

- Documents containing "the" n=4

Calculating IDF for "the": $IDF(\text{ the }) = \log(5/4) \approx 0.0969$

This shows that "cat" has a higher IDF score than "the," reflecting its rarity.

TF-IDF Vectorization

TF-IDF combines both TF and IDF to provide a balanced measure that reflects both the importance of a term within a specific document and its rarity across all documents.

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$$

TF-IDF effectively balances the frequency of terms within individual documents against their overall rarity across a corpus, making it an invaluable tool for information retrieval and text classification tasks. By using both metrics together, it helps highlight significant terms that can improve model predictions and enhance understanding of textual data.

TF-IDF Vectorizer

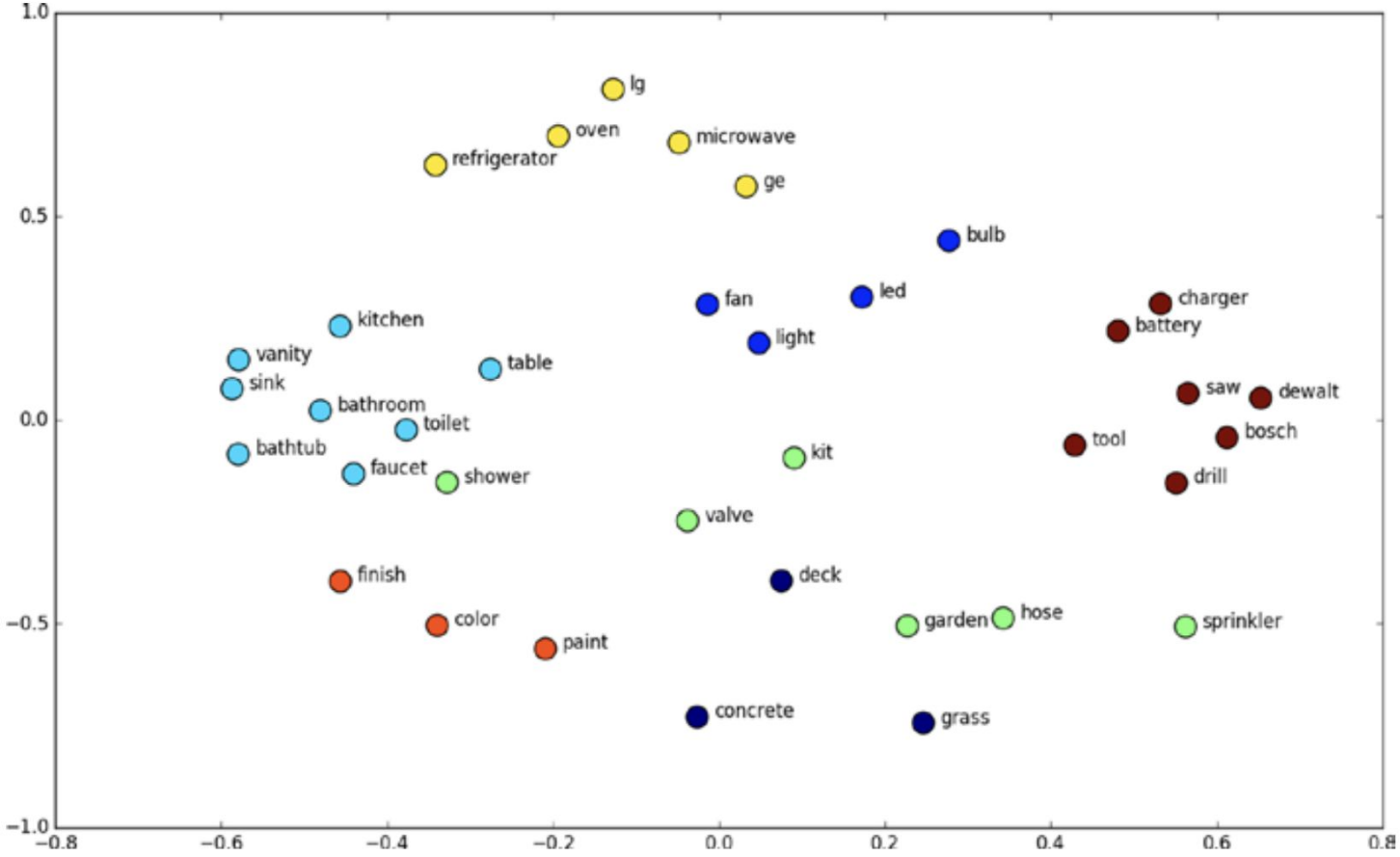
```
from sklearn.feature_extraction.text import TfidfVectorizer

# Step 1: Prepare the text data
text = [
   "The quick brown fox jumped over the lazy dog.",
   "The dog.",
   "The fox"
]

# Step 2: Create TF-IDF Vectorizer
vectorizer = TfidfVectorizer()

# Step 3: Fit the vectorizer to the text (build vocabulary)
vectorizer.fit(text)

# Step 4: Print vocabulary
print("Vocabulary:")
print(vectorizer.vocabulary_)

# Step 5: Print IDF values
print("IDF Values:")
print(vectorizer.idf_)

# Step 6: Transform the text to TF-IDF features
tfidf_matrix = vectorizer.transform(text)

# Step 7: Print the TF-IDF matrix
print("TF-IDF Matrix:")
print(tfidf_matrix.toarray())

# Step 8: Get feature names to understand the matrix
print("Feature Names:")
print(vectorizer.get_feature_names_out())

# Vocabulary:
# {'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5, 'lazy': 4, 'dog': 1}
# IDF Values:
# [1.69314718 1.28768207 1.28768207 1.69314718 1.69314718 1.69314718 1.69314718 1.        ]
# TF-IDF Matrix:
# [[0.36388646 0.27674503 0.27674503 0.36388646 0.36388646 0.36388646 0.36388646 0.42983441]
#  [0.         0.78980693 0.         0.         0.         0.         0.         0.61335554]
#  [0.         0.         0.78980693 0.         0.         0.         0.         0.61335554]]
# Feature Names:
# ['brown' 'dog' 'fox' 'jumped' 'lazy' 'over' 'quick' 'the']
```

Word Embeddings

- Word embeddings are the dense vector representations of words in a continuous vector space.
- Words with similar meanings cluster together in this space
- All other methods like TF-IDF, One Hot Encoder, etc. are based on frequency and hence called frequency-based embeddings or features. And, prediction-based embeddings are called word embeddings.
- Problem with frequency approaches:
  - I am eating an apple.
  - I am using apple.
- Word embedding is the feature learning technique where words from the vocabulary are mapped to vectors of real numbers capturing the contextual hierarchy.

Word Embeddings

king + woman − man → queen



Word Embeddings

- Word embeddings are prediction based, and they use shallow neural networks to train the model that will lead to learning the weight and using them as a vector representation.
- Some word embedding approaches
  - Word2Vec (Skip-Gram, Continuous Bag of Words (CBOW)) - (2013 - by Tomas Mikolov et al. at Google)
  - FastText (character n-grams) - (2016 (by Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov at Facebook AI Research)
  - Glove (matrix factorization technique on word context matrix) - (2014 by Jeffrey Pennington, Richard Socher, and Christopher Manning at Stanford)

Word2vec

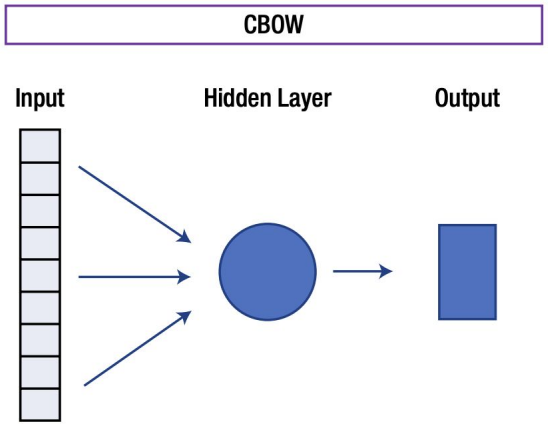
Word2vec is the deep learning Google framework to train word embeddings. It will use all the words of the whole corpus and predict the nearby words. It will create a vector for all the words present in the corpus in a way so that the context is captured. It also outperforms any other methodologies in the space of word similarity and word analogies.

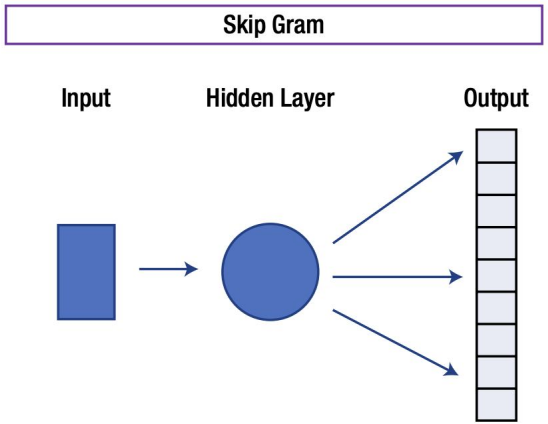
https://arxiv.org/pdf/1301.3781

https://arxiv.org/pdf/1310.4546

https://en.wikipedia.org/wiki/Word2vec

Word2vec Approaches





Example: Consider the sentence: "The cat sits on the mat"

CBOW scenario:

- Target word: "sits"
- Context words: "The", "cat", "on", "the"
- Model averages context word vectors to predict "sits"

Skip-gram scenario:

- Target word: "sits"
- Predicts probable context words like "cat", "on"

Predicts a target word given its context words

Architecture:

- Input: Multiple context words (averaged)
- Output: Probability distribution for target word

Characteristics:

- Faster training than Skip-gram
- Better with frequent words
- Smoother embeddings due to averaging

Predicts context words given a target word

Architecture:

- Input: Single target word
- Output: Probability distribution for context words

Characteristics:

- Better with rare words
- More computationally intensive
- Often performs better on semantic tasks

Training Word2Vec

FYI

Architectures:

Continuous Bag of Words (CBOW) and Skip-gram.

The training process involves several key steps:

1. Preprocessing the Text Data
   Tokenization, Lowercasing, Removing stopwords, punctuation, and special characters (optional), Building a vocabulary, Handling rare words
2. Creating Training Data
   Word2Vec learns relationships between words by using a sliding window technique:
   - CBOW: Predicts the target word based on surrounding context words.
   - Skip-gram: Predicts surrounding words based on a given target word.
   - The size of the sliding window (context window) affects how much context is considered.
3. Initializing Word Vectors
   - Word vectors are initialized randomly.
   - During training, the model adjusts these vectors to minimize prediction errors.
4. Training Using a Neural Network
   The model uses a simple neural network with:
   - An input layer representing one-hot encoded words.
   - A hidden layer (projection layer) where words are mapped into a lower-dimensional vector space.
   - An output layer using a softmax function to predict target/context words.

   The objective function is to maximize the probability of predicting the correct word given the context.
5. Using Negative Sampling or Hierarchical Softmax
   Since computing softmax over a large vocabulary is expensive, Word2Vec uses:
   - Negative Sampling: Only updates a small set of incorrect (negative) words instead of the entire vocabulary.
   - Hierarchical Softmax: Organizes words in a tree structure to speed up probability calculations.
6. Training Iterations (Epochs)
   - The model trains over multiple passes (epochs) through the dataset.
   - Words appearing frequently get refined faster.
7. Saving and Using Word Embeddings
   - Once trained, the word embeddings (word vectors) are stored.
   - These embeddings can be used for similarity searches, NLP models, clustering, or semantic analysis.

Word2vec

```
# !pip install gensim
from gensim.models import Word2Vec
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Example sentences
sentences = [
    ['I', 'love', 'nlp'],
    ['I', 'will', 'learn', 'nlp', 'in', '2', 'months'],
    ['nlp', 'is', 'future'],
    ['nlp', 'saves', 'time', 'and', 'solves', 'lot', 'of', 'industry', 'problems'],
    ['nlp', 'uses', 'machine', 'learning']
]

# Training the model using CBOW (Continuous Bag of Words)
model = Word2Vec(sentences, vector_size=50, window=3, min_count=1,
sg=0)  # sg=0 for CBOW, sg=1 for Skip-Gram

# Print model information
print("Model Information:")
print(model)

# Access vector for 'nlp'
print("Vector for 'nlp':")
print(model.wv['nlp'])

# Save model
model.save('cbow.bin')
```

```
# Use the model

# Load saved model
model = Word2Vec.load('cbow.bin')
vocab = list(model.wv.index_to_key)
print("Vocab", vocab)

word1 = "learn"
word2 = "solves"
vector1 = model.wv[word1]
vector2 = model.wv[word2]
print(word1, vector1)
print(word2, vector2)

sim = model.wv.similarity(word1, word2)
print(f"Similarity({word1}, {word2}):", sim)
```

Word2vec

```
# Visualize embeddings
# Apply PCA for dimensionality reduction
pca = PCA(n_components=2)
model = Word2Vec.load('cbow.bin')
vocab = list(model.wv.index_to_key)
print("Vocab", vocab)
vectors = [model.wv[word] for word in vocab]
result = pca.fit_transform(vectors)

# Create scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(result[:, 0], result[:, 1])

# Annotate points with words
for i, word in enumerate(vocab):
   plt.annotate(word, xy=(result[i, 0], result[i, 1]))

plt.title('Word Embeddings Visualization (CBOW)')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.tight_layout()
plt.show()
```

fastText

fastText is another deep learning framework developed by Facebook to capture context and meaning. fastText is the improvised version of word2vec. word2vec basically considers words to build the representation. But fastText takes each character while computing the representation of the word.

https://fasttext.cc/

https://github.com/facebookresearch/fastText

https://en.wikipedia.org/wiki/FastText

fastText

```
from gensim.models import FastText

# Example sentences
sentences = [
   ['I', 'love', 'nlp'],
   ['I', 'will', 'learn', 'nlp', 'in', '2', 'months'],
   ['nlp', 'is', 'future'],
   ['nlp', 'saves', 'time', 'and', 'solves', 'lot', 'of', 'industry',
'problems'],
   ['nlp', 'uses', 'machine', 'learning']
]

# Train FastText model
fast = FastText(sentences, vector_size=20, window=1,
min_count=1, workers=5, min_n=1, max_n=2)

# Demonstrate vector retrieval for words
print("Vector for 'nlp':")
print(fast.wv['nlp'])

print("Vector for 'deep':")
print(fast.wv['deep'])

# Save the model
fast.save('fast.bin')
```

```
# Use the model
# Load saved model
model = FastText.load('fast.bin')
vocab = list(model.wv.index_to_key)
print("Vocab", vocab)

word1 = "learn"
word2 = "solves"
vector1 = model.wv[word1]
vector2 = model.wv[word2]
print(word1, vector1)
print(word2, vector2)

sim = model.wv.similarity(word1, word2)
print(f"Similarity({word1}, {word2}):", sim)

# Demonstrate FastText's ability to generate vectors for out-of-vocabulary
words
print("Demonstrating out-of-vocabulary word vector generation:")
out_of_vocab_words = ['learning', 'deeplearning', 'nlpmodel']
for word in out_of_vocab_words:
   print(f"Vector for '{word}': {fast.wv[word]}")
```

GloVe (Global Vectors for Word Representation)

- GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.
- Uses **matrix factorization** (co-occurrence matrix) rather than a neural network.
- Captures both **global and local** word relationships.
- Typically results in more **interpretable embeddings** compared to W2V.
- Commonly used for NLP tasks requiring high-quality pre-trained embeddings.

https://en.wikipedia.org/wiki/GloVe
https://nlp.stanford.edu/projects/glove/
https://nlp.stanford.edu/pubs/glove.pdf

Glove

```
# Download pre-trained glove model
#
https://huggingface.co/stanfordnlp/glove/resolve/main/glove.6
B.zip
# Load using custom method
def load_glove_embeddings(glove_model_path):
    embeddings_index = {}
    with open(glove_model_path, "r", encoding="utf-8") as f:
        for line in f:
            values = line.strip().split()
            word = values[0]
            vector = np.asarray(values[1:], dtype=np.float32)
            embeddings_index[word] = vector
    return embeddings_index


glove_model_path = "../../data/glove.6B.50d.txt"
model = load_glove_embeddings(glove_model_path)

# Access word vector
print(model["hello"])
```

```
# Load using gensim
from gensim.models import KeyedVectors

glove_model_file = "../../data/glove.6B.50d.txt"

# Load with gensim
model =
KeyedVectors.load_word2vec_format(glove_model_file,
binary=False, no_header=True)

print(model["hello"])
```

Glove

```
# Use the model
# Load saved model
vocab = model.index_to_key[:100]
print(f"Vocab (100/{len(model.index_to_key)})", vocab)

word1 = "learn"
word2 = "solves"
vector1 = model[word1]
vector2 = model[word2]
print(word1, vector1)
print(word2, vector2)

sim = model.similarity(word1, word2)
print(f"Similarity({word1}, {word2}):", sim)
```

```
# king - man + women ~= queen
v1 = model.get_vector("king")
v2 = model.get_vector("man")
v3 = model.get_vector("women")
v4 = model.get_vector("queen")
result = v1-v2+v3

print(result)
print(v4)

sim = model.similar_by_vector(result)
print(sim)

sim4 = model.similar_by_vector(v4)
print(sim4)
```

SMS Spam Classification

FYI

```
# Download SMS spam-ham dataset from here:
https://www.kaggle.com/datasets/uciml/sms-spam-collection-dataset

import nltk
import pandas as pd
import sklearn
import textblob

# Step 1. Data collection
# Read the data
df = pd.read_csv("../data/spam.csv", encoding='latin1')
# Data understanding
print(df.columns)
# Output
df = df[['v1', 'v2']]
df = df.rename(columns={"v1": "target", "v2": "email"})
print(df.head())


# Step 2. Text processing and feature engineering
# Text pre-processing steps like lower case, stemming and lemmatization
df['email'] = df['email'].apply(lambda x: " ".join(x.lower() for x in x.split()))
stop = nltk.corpus.stopwords.words('english')
df['email'] = df['email'].apply(lambda x: " ".join(x for x in x.split() if x not in stop))
st = nltk.stem.PorterStemmer()
df['email'] = df['email'].apply(lambda x: " ".join([st.stem(word) for word in x.split()]))
df['email'] = df['email'].apply(lambda x: " ".join([textblob.Word(word).lemmatize() for
word in x.split()]))
print(df.head())
```

```
# Splitting data into train and validation
train_x, test_x, train_y, test_y = sklearn.model_selection.train_test_split(df['email'], df['target'])

# TFIDF feature generation for a maximum of 5000 features
encoder = sklearn.preprocessing.LabelEncoder()
train_y = encoder.fit_transform(train_y)
test_y = encoder.fit_transform(test_y)
tfidf_vect = sklearn.feature_extraction.text.TfidfVectorizer(analyzer='word',
token_pattern=r'\w{1,}',
                                                    max_features=5000)
tfidf_vect.fit(df['email'])
train_x_tfidf = tfidf_vect.transform(train_x)
test_x_tfidf = tfidf_vect.transform(test_x)
print(train_x_tfidf.data)

Step 3. Model training
def train_model(classifier, train_x, train_y, test_x, test_y, is_neural_net=False):
  # fit the training dataset on the classifier
  classifier.fit(train_x, train_y)
  # predict the labels on validation dataset
  predictions = classifier.predict(test_x)
  return sklearn.metrics.accuracy_score(predictions, test_y)

# Naive Bayes training
accuracy = train_model(sklearn.naive_bayes.MultinomialNB(alpha=0.2), train_x_tfidf, train_y,
test_x_tfidf, test_y)
print("Accuracy: ", accuracy)

# Linear Classifier on Word Level TF IDF Vectors
accuracy = train_model(sklearn.linear_model.LogisticRegression(), train_x_tfidf, train_y,
test_x_tfidf, test_y)
print("Accuracy: ", accuracy)
```

Sequence-to-Sequence Language Learning: RNNs

Sequence-to-Sequence Models

- A Sequence-to-Sequence (Seq2Seq) model is designed to take a sequence of data (e.g., a sentence) as input and transform it into another sequence (e.g., a translated sentence). Seq2Seq is commonly used for tasks like machine translation, summarization, and speech-to-text.

- Sequence learning is fundamental for NLP tasks where both input and output data are sequences, and each input token may depend on previous tokens.

Applications:

- Machine Translation - For example, translating English text to Nepali.
- Speech Recognition - Converting spoken language into written form.
- Text Summarization - Producing a summary of a longer text.
- Chatbots - Generating conversational responses.

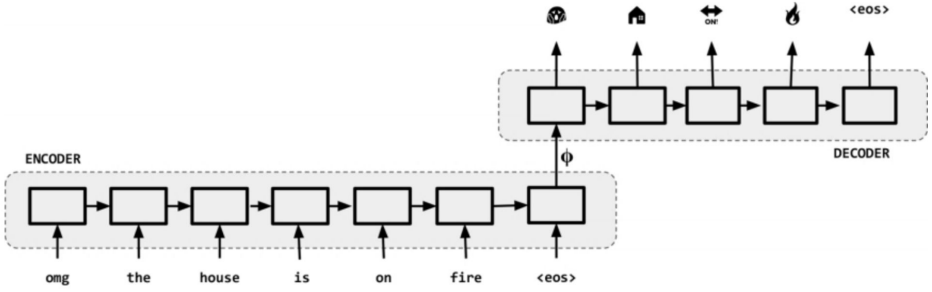


Figure 8-4. An S2S model for translating English to emoji.

Traditional Methods for Sequence Learning

- **Rule-Based Systems:**
  - Systems that use manually defined rules (e.g., for translation). These systems struggle with variability in natural language and are not scalable.

- **Statistical Methods:**
  - ***Hidden Markov Models (HMMs):*** Used for sequence prediction tasks but are limited by their assumptions.

  - ***N-Gram Models:*** Statistical models that predict the next word based on the previous "n" words, but they do not handle long dependencies well.

- **Limitations:**
  - These traditional methods often fail to capture long-term dependencies and fail to generalize to unseen data.

Recurrent Neural Networks (RNNs)

- RNNs are neural networks designed for sequence data, with loops that allow information to be passed from one step to the next.
- RNNs process input sequences one element at a time, maintaining a hidden state that encodes the sequence's history.

- The hidden state is updated at each time step based on the input and the previous hidden state.

Recurrent Neural Networks(RNN)

The previous step's hidden layer and final outputs are fed back into the network and will be used as input to the next steps' hidden layer, which means the network will remember the past and it will repeatedly predict what will happen next. The drawback in the general RNN architecture is that it can be memory heavy, and hard to train for long term temporal dependency (i.e., context of long text should be known at any given stage).

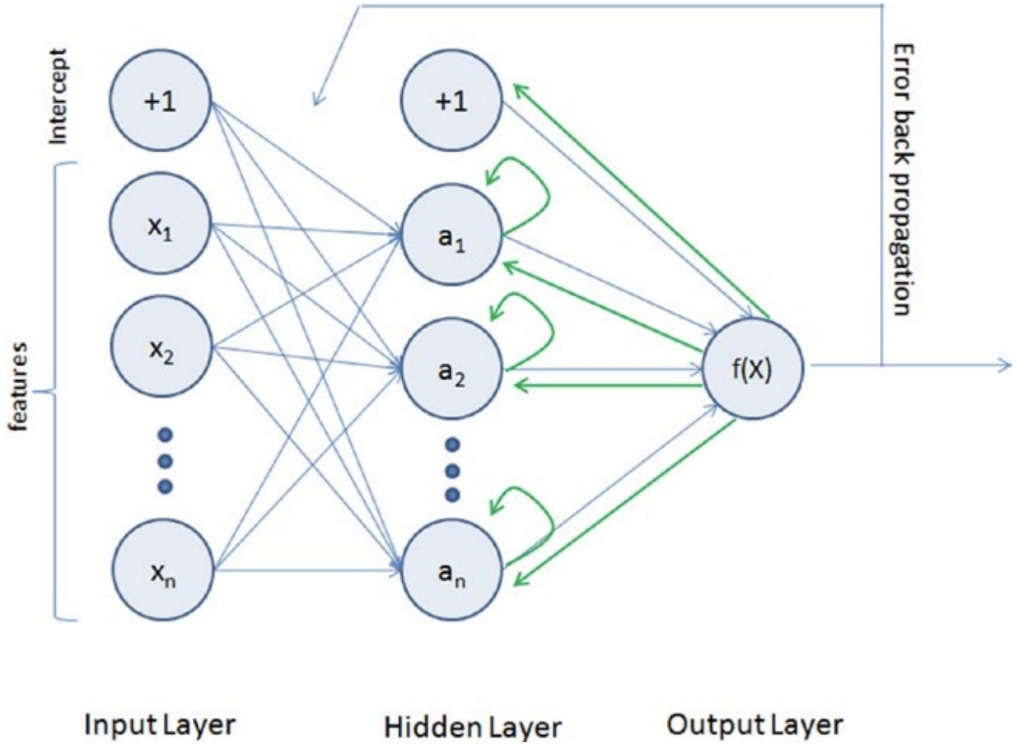
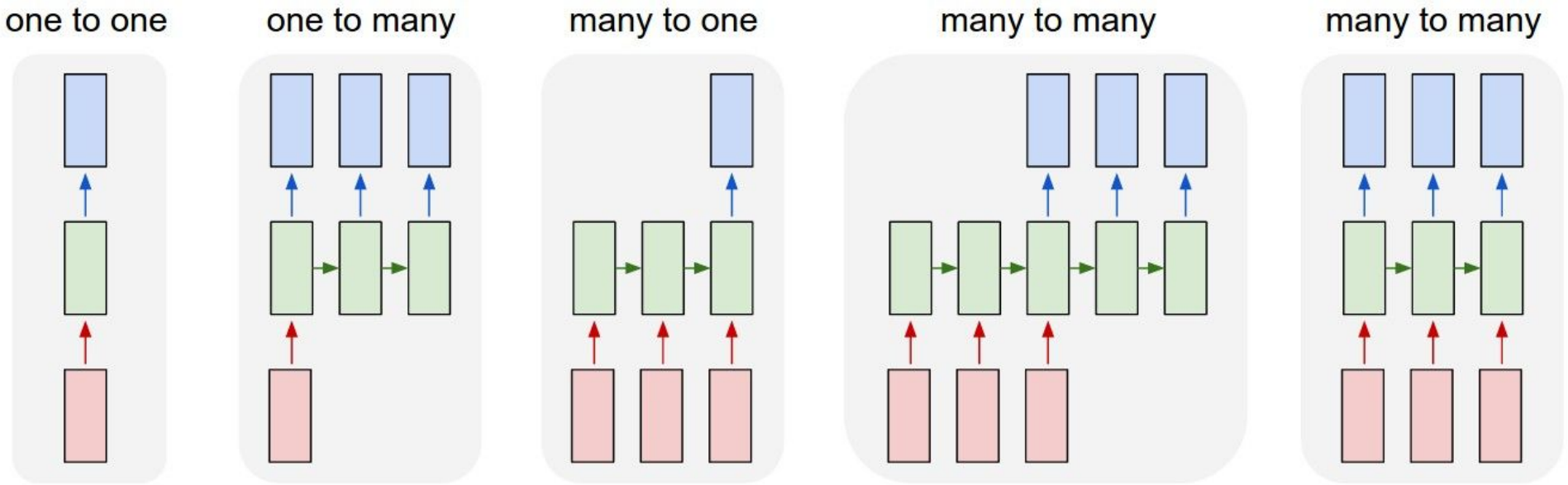


Figure 6-7. *Recurrent Neural Network*

RNN Sequence Modelling



Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Architecture of an RNN

The basic unit is a recurrent neuron that processes one token at a time, updates its hidden state, and passes this updated state to the next token in the sequence.

Forward Pass:

- The input is passed through the RNN layer, which updates the hidden state at each time step and produces an output.

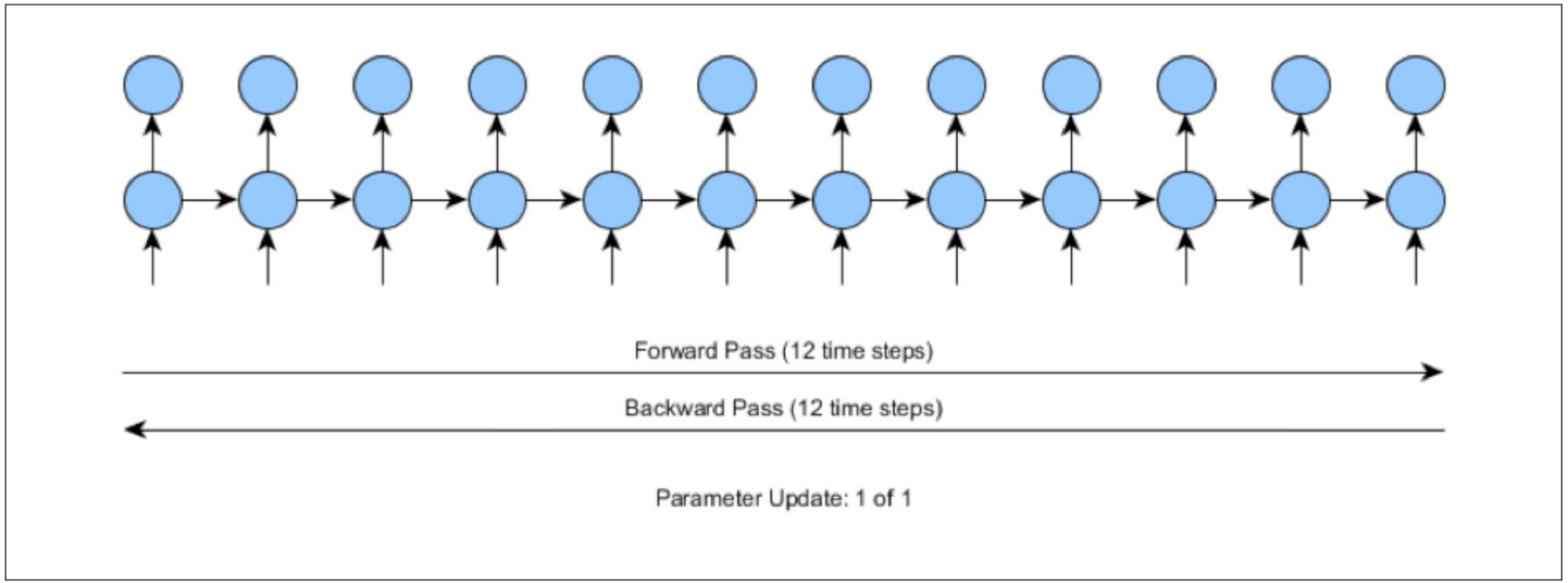
Backpropagation Through Time (BPTT):

- A form of backpropagation applied to sequences, which unrolls the RNN over time and adjusts the weights by computing the gradient of the loss function.

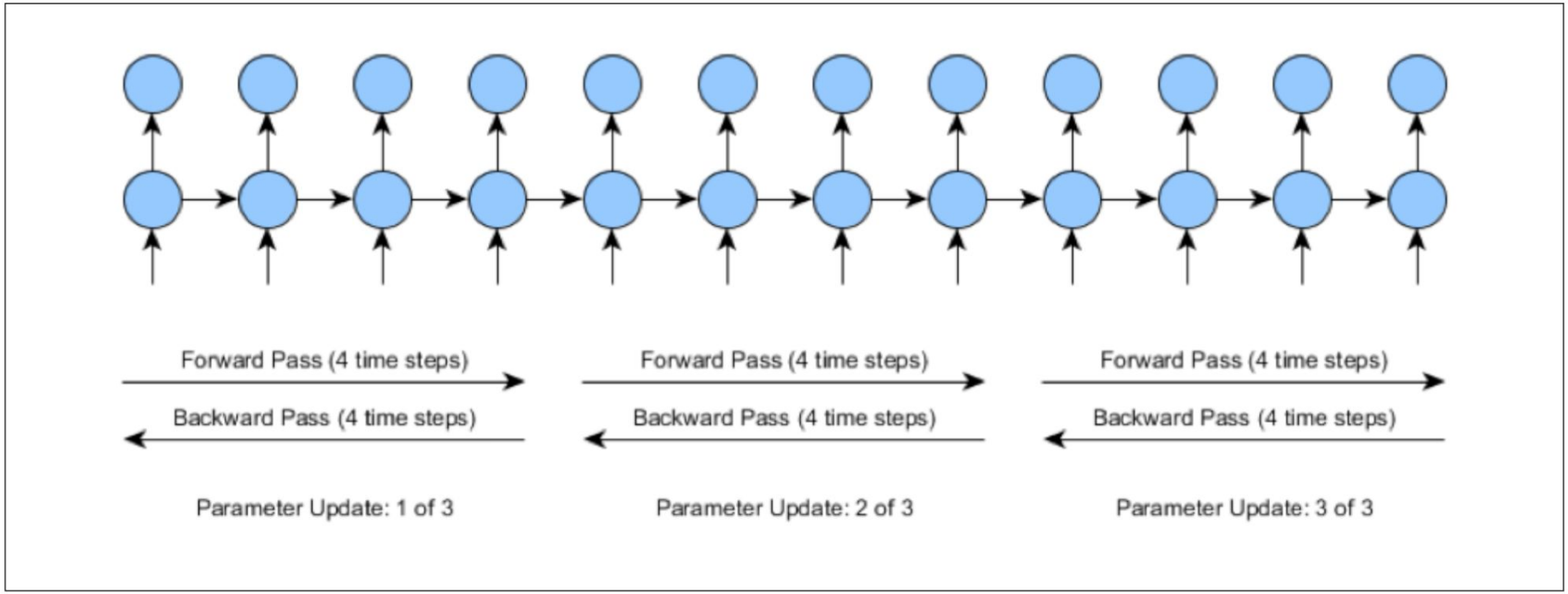
Backpropagation Through Time (BPTT)

- Backpropagation Through Time (BPTT)
  - BPTT is a variant of backpropagation used for training Recurrent Neural Networks.
  - It applies the chain rule to compute gradients based on the network's connection structure, allowing error signals to flow backward through time from future time-steps to current ones.
- Truncated BPTT
  - Traditional BPTT can be computationally expensive, especially with long sequences.
  - Truncated BPTT is a technique used to reduce the computational complexity of parameter updates in RNNs.
  - It involves breaking long sequences into shorter segments, allowing for more frequent parameter updates and faster training.
  - Truncated BPTT is recommended for sequences with more than a few hundred time-steps to improve training efficiency.

Backpropagation Through Time (BPTT)



Truncated BPTT



Recurrent Neural Networks (RNNs)

RNN Challenges

- Vanishing Gradient Problem: Gradients can diminish as they are propagated through many time steps, making it hard to learn long-range dependencies.
- Exploding Gradients: Large gradients can cause unstable model training.

Seq2Seq Task Challenges

- Variable Length Sequences: Input and output sequences vary in length. Traditional RNNs struggle to handle this efficiently.
- Long-Term Dependencies: In tasks like translation, earlier words in a sentence influence the meaning of later words, requiring models to learn long-term dependencies.
- Contextual Understanding: Capturing the context of an entire sequence to generate coherent outputs is challenging.

Vanishing Gradient Problem

The vanishing gradient problem occurs when the gradients of the loss function become very small, approaching zero. This leads to minimal updates to the weights of the network during training, effectively halting learning.

Causes

- **Activation Functions:** Certain activation functions, like sigmoid and tanh, produce small gradients in their inactive regions. As these small gradients propagate backward through many layers, they can diminish to near-zero values.
- **Network Depth:** In deep networks, as gradients are back propagated through multiple layers, they can shrink exponentially, making it difficult for the model to learn long-range dependencies

Consequences

- **Slow Learning:** The network struggles to learn from distant inputs, leading to poor performance on tasks that require understanding of long-term dependencies.
- **Underfitting:** The model fails to capture the complexity of the data, resulting in low accuracy and generalization capability

Exploding Gradient Problem

The exploding gradient problem occurs when gradients grow exponentially during backpropagation, leading to extremely large updates to the network weights. This can cause numerical instability and make the model unable to converge.

Causes

- **Activation Functions:** Using activation functions that yield large gradients (e.g., ReLU with high learning rates) can lead to rapid growth of gradients as they propagate through layers.
- **Weight Initialization:** Poor initialization of weights can exacerbate this issue; if weights are initialized too large, it can lead to rapid increases in gradient values.

Consequences

- **Instability:** The model may become unstable and erratic in its predictions, often resulting in NaN (not a number) values for weights during training.
- **Overfitting:** The model may learn to fit noise in the training data rather than generalizable patterns due to excessive weight updates

Solution to Vanishing and Exploding Gradient Problem

1. **Gradient Clipping:** This technique involves setting a threshold for gradients; if they exceed this threshold, they are scaled down. This helps prevent gradients from becoming too large during backpropagation.
2. **Weight Initialization:** Proper initialization methods like Xavier or He initialization can help maintain stable gradients throughout training.
3. **Use of Advanced Architectures:**
    - Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) are designed with mechanisms that help manage gradients effectively, allowing them to capture long-term dependencies without suffering as severely from these problems.
4. **Batch Normalization:** Applying normalization techniques can help stabilize learning by ensuring that the inputs to each layer maintain a consistent distribution.
5. **Skip Connections:** These connections allow gradients to bypass certain layers, which can help maintain their magnitude and prevent them from vanishing.

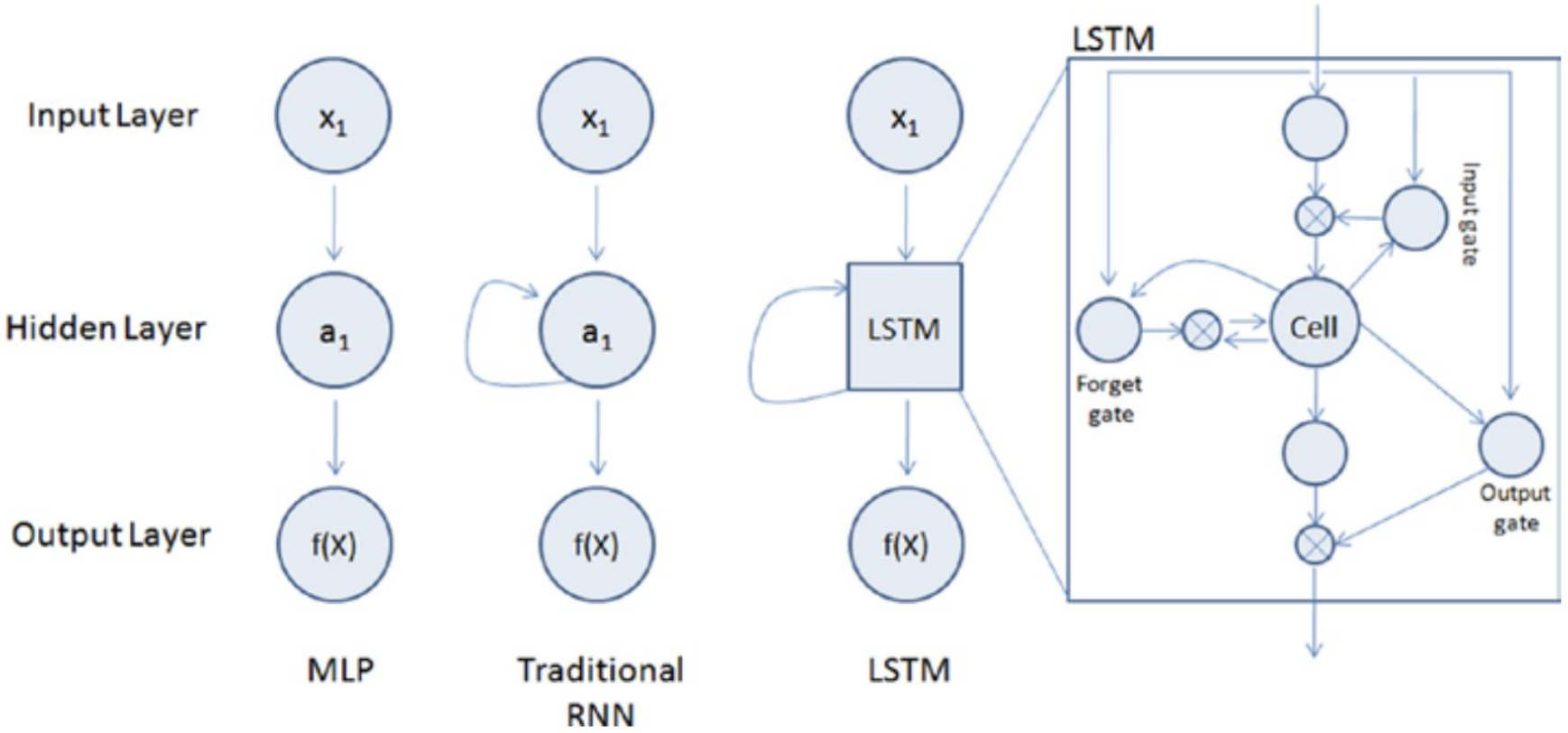
Long Short-term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a specialized type of Recurrent Neural Network (RNN) designed to address the challenges of learning long-term dependencies within sequential data. Introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997, LSTMs have become a foundational architecture in deep learning, particularly for tasks involving time series, natural language processing, and speech recognition.

LSTM address the vanishing gradient problem commonly found in traditional RNNs.

LSTMs enable the modeling of long-range dependencies in sequential data through a sophisticated architecture that includes memory cells and gating mechanisms.

Long Short-term Memory (LSTM)



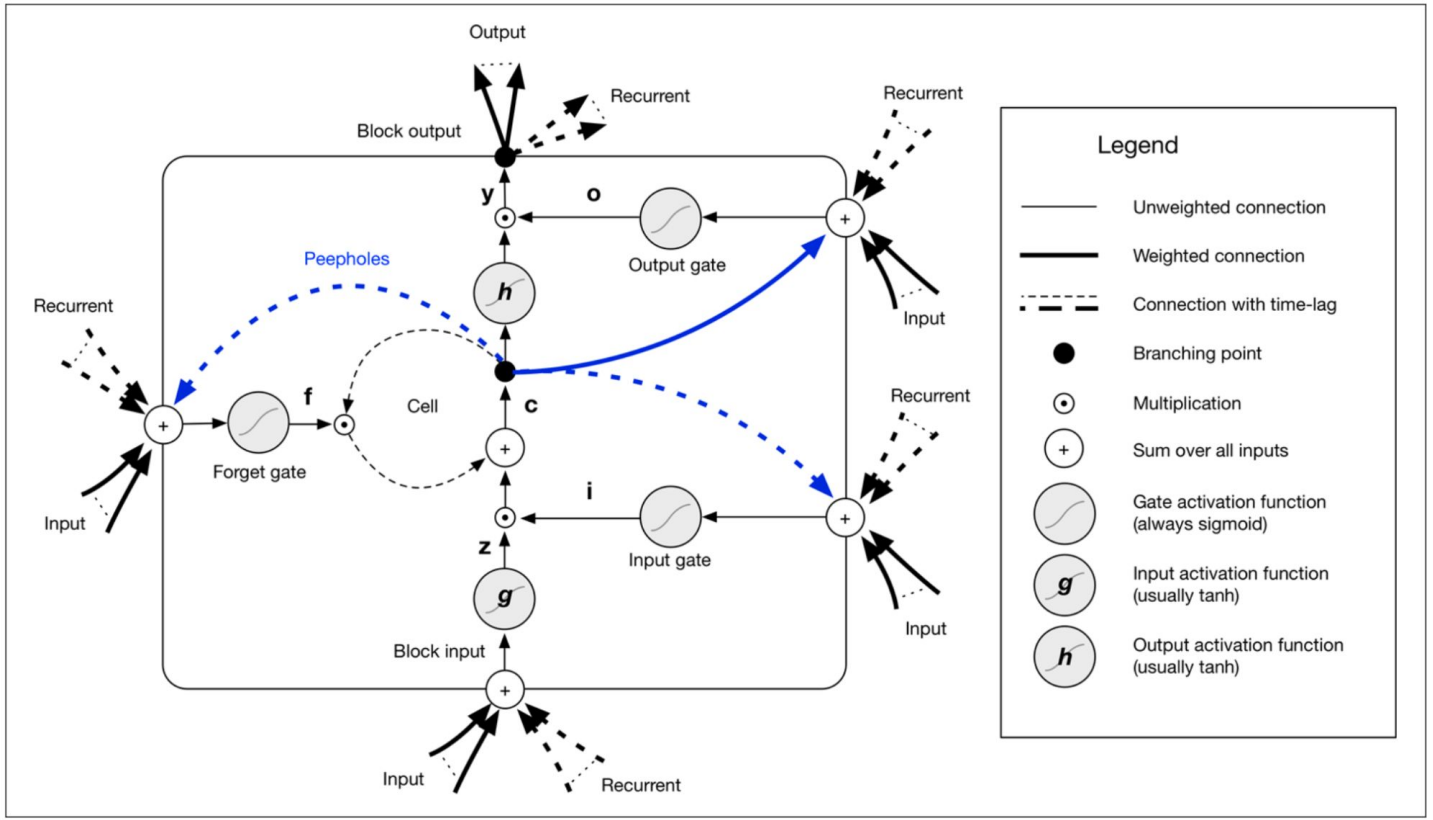
LSTM Units

The units in the layers of Recurrent Neural Networks are a variation on the classic artificial neuron.

Each LSTM unit has two types of connections:

- Connections from the previous time-step (outputs of those units)
- Connections from the previous layer

The memory cell in an LSTM network is the central concept that allows the network to maintain state over time. The main body of the LSTM unit is referred to as the LSTM block



Long Short-term Memory (LSTM)

Memory Cells

LSTMs utilize memory cells that can maintain information over long periods. This capability allows the network to remember important data while discarding irrelevant information.

Gating Mechanisms - LSTMs employ three types of gates:

- Input Gate: Controls how much new information is added to the memory cell.
- Forget Gate: Determines what information should be discarded from the memory cell.
- Output Gate: Decides what information from the memory cell should be output to the next layer.

These gates use sigmoid activation functions, which output values between 0 and 1, allowing for smooth control over the information flow.

No Activation Function in Recurrent Components

Unlike traditional RNNs, LSTMs do not apply activation functions within their recurrent components, which helps prevent gradients from vanishing during backpropagation.

Long Short-term Memory (LSTM)

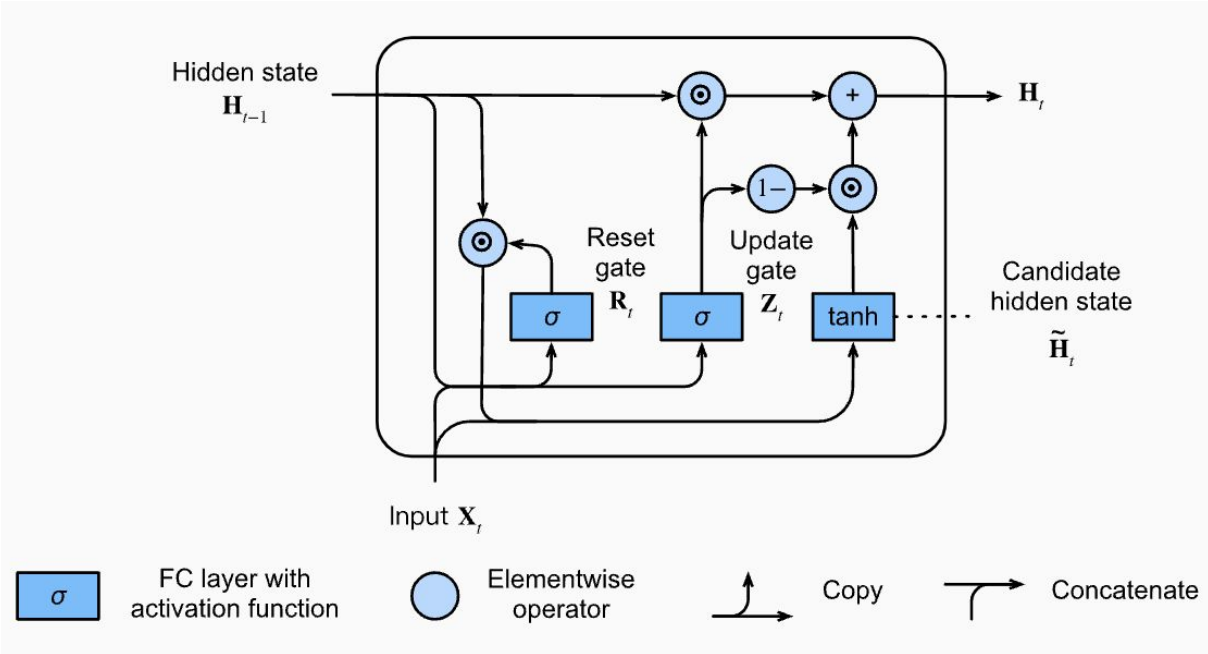
LSTM Component	Formula
Input gate layer: This decides which values to store in the cell state.	$i_t = \text{sigmoid}(w_i x_t + u_i h_{t-1} + b_i)$
Forget gate layer: As the name suggested this decides what information to throw away from the cell state.	$f_t = \text{sigmoid}(W_f x_t + U_f h_{t-1} + b_f)$
Output gate layer: Create a vector of values that can be added to the cell state.	$O_t = \text{sigmoid}(W_o x_t + u_i h_{t-1} + b_o)$
Memory cell state vector.	$c_t = f_t \, o \, c_{t-1} + i_t o * \text{hyperbolic tangent}(W_c x_t + u_c h_{t-1} + b_c)$

Gated Recurrent Unit Networks(GRU)

Gated Recurrent Units (GRUs) are a type of Recurrent Neural Network (RNN) architecture designed to handle sequential data more effectively than traditional RNNs. Introduced by Kyunghyun Cho et al. in 2014, GRUs address the vanishing gradient problem that often hampers the performance of standard RNNs, especially when dealing with long sequences of data.

GRUs were introduced as a variant of the LSTM (Long Short-Term Memory) network, designed to address some of the complexities and computational inefficiencies in LSTMs.

Gated Recurrent Unit Networks(GRU)



$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{\text{xr}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hr}} + \mathbf{b}_{\text{r}}),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{\text{xz}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hz}} + \mathbf{b}_{\text{z}}),$$

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{\text{xh}} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}),$$

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

Gated Recurrent Unit Networks(GRU)

GRU has two main gates:

- **Update Gate:** Decides how much of the past information needs to be passed forward to the future.
- **Reset Gate:** Controls how much of the previous information to forget.

GRUs combine the cell state and hidden state into a single vector, unlike LSTMs that maintain separate vectors for these. This simplification results in fewer parameters, making GRUs computationally less expensive than LSTMs.

GRUs offer several advantages, including **fewer parameters for faster training**, a **simpler architecture** for easier implementation, **comparable performance to Long LSTM** networks, **effective management of long-term dependencies**, and greater **computational efficiency** due to their reduced complexity

Building Seq2Seq Models

Encoder-Decoder Architecture

- Encoder
  - The encoder processes the input sequence, generating a fixed-length context vector that summarizes the entire input sequence.
  - In an RNN-based Seq2Seq model, the encoder passes the final hidden state as the context.
- Decoder
  - The decoder uses the context vector and generates the output sequence, one token at a time.
  - In a simple model, the decoder starts with the context vector and generates the output step by step.
- Application Example:
  - For machine translation, the encoder takes an English sentence, and the decoder outputs the corresponding sentence in Nepali.

Training and Optimizing Seq2Seq Models

- Loss Function
  - The most common loss function used for Seq2Seq tasks is Cross-Entropy Loss, which measures the difference between predicted and actual tokens.

- Training Strategies
  - Teacher Forcing: During training, the true output token from the previous time step is fed into the decoder as the next input, improving convergence speed.
  - Scheduled Sampling: Gradually shifts from using true tokens to the predicted tokens during training.

- Optimizers
  - Adam optimizer is commonly used for Seq2Seq models, as it adjusts learning rates based on gradients to accelerate convergence.

Applications of Encoder-Decoder Models in NLP

- Machine Translation: Translating text from one language to another (e.g., Google Translate).
- Text Summarization: Generating concise summaries from longer texts.
- Question Answering: Providing relevant answers to user queries based on context.
- Language Modeling: Predicting the next word in a sequence, used in applications like autocomplete and chatbots.
- Seq2Seq for Speech Recognition: Converts audio signals into text by treating the audio as an input sequence and the text as the output.
- Text-to-Speech: The inverse process, generating human-like speech from text.

Thank You