EE7207 Lecture 8

About me

Just call me Nick!

Nick LUO Wuqiong
Data Science Director
NLP and Generative AI
American Express

Foundations: Recap on Neural Networks

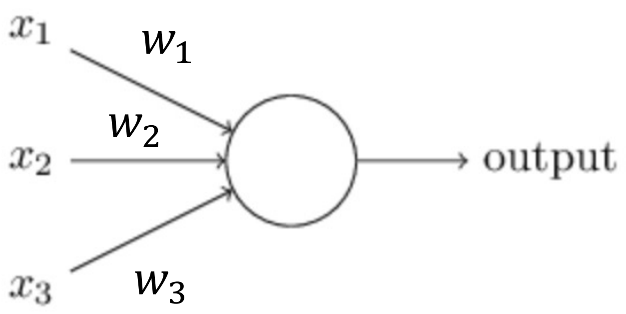
Neurons

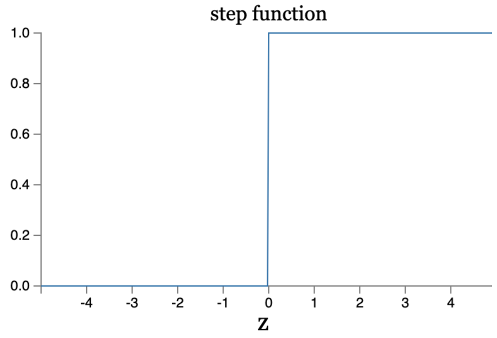
A neuron performs the following operations:

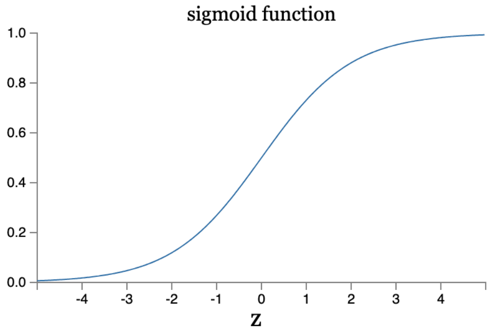
1. **Takes Inputs:**
   - Each neuron receives inputs $(x_1, x_2, \ldots, x_n)$
   - Each input is associated with a weight $(w_1, w_2, \ldots, w_n)$ that determines its importance.
2. **Computes a Weighted Sum:**
   The neuron calculates a weighted sum of the inputs:

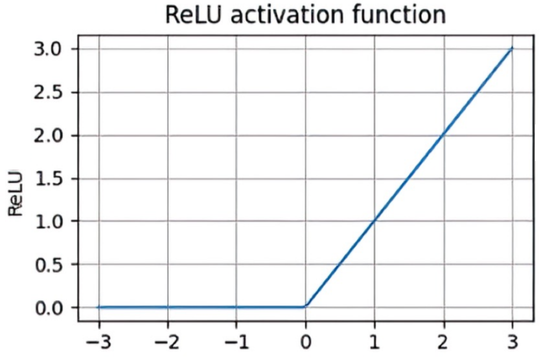
$$z = \sum_{i=1}^{n} w_i \cdot x_i + b$$

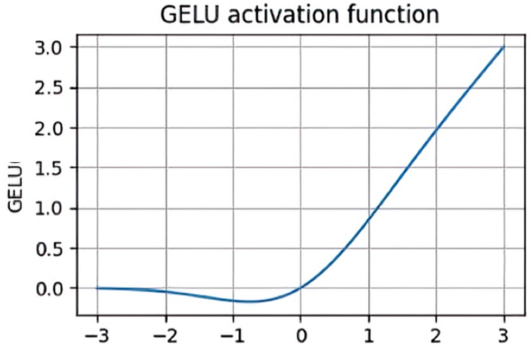
   Here, b is the **bias**, which helps the model shift the activation functior and allows it to learn better.
3. **Applies an Activation Function:**
   - The weighted sum (z) is passed through an **activation function** to introduce **non-linearity**.
   - Common activation functions include Sigmoid and ReLU (Rectified Linear Unit).
4. **Produces an Output:**
   - The output of the activation function is passed to the next layer of neurons or used as the final output of the network.











Why Non-Linearity is Needed in Neural Networks?

If we do not use a **non-linear activation function**, all layers in the network (input, hidden, and output) essentially collapse into a single equivalent layer. This happens because a composition of linear functions is still a linear function.

For example:

$$f(g(h(x))) = w_3 \cdot (w_2 \cdot (w_1 \cdot x + b_1) + b_2) + b_3$$

This simplifies into:

$$f(x) = w' \cdot x + b'$$

where w' and b' are the combined weights and biases.

Neural networks with at least one hidden layer, non-linear activation functions, and sufficient neurons can approximate any continuous function. This is known as the **Universal Approximation Theorem**.

Neuron Networks

Neuron networks consist of layers of interconnected **neurons** that process and transform input data to produce meaningful outputs.

A typical neural network is organized into three types of layers:

Input Layer

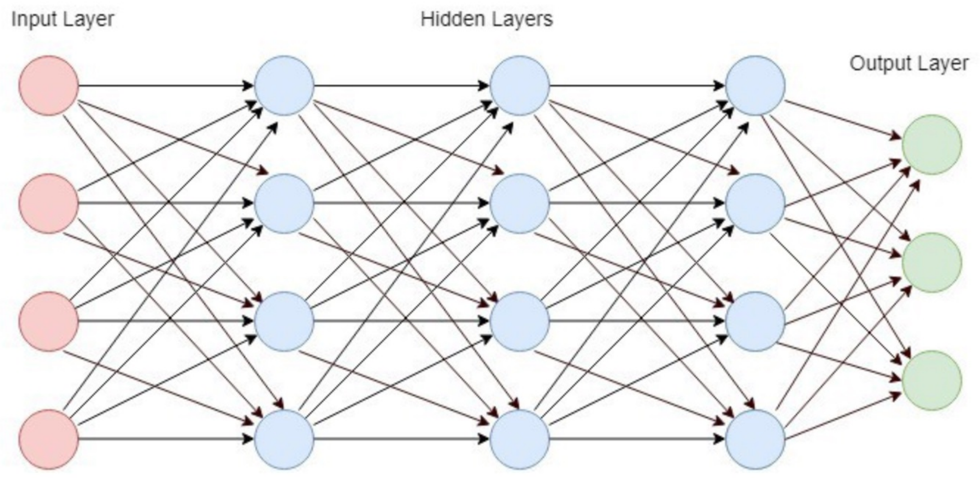
- Accepts raw data as input to the network.

Hidden Layers

- Perform computations and extract patterns/features from the input data.
- These layers are where the network learns from the data.
- Hidden layers uncover complex relationships that are not explicitly provided in the input data.

Output Layer

- Produces the final result of the network's computations.



Forward Propagation

Loss

For training data, we are given **input-output pairs** (x, y), where:

- x: Input data (e.g., historical stock prices).
- y: Expected output (e.g., next day stock price).

During **forward propagation**, the neural network processes the input x and produces a **prediction** $\hat{y}$.

Loss measures the difference between the actual expected output y and the predicted output $\hat{y}$.

Purpose of Loss:

- To quantify how well the model's predictions align with the ground truth.
- A smaller loss indicates better performance.

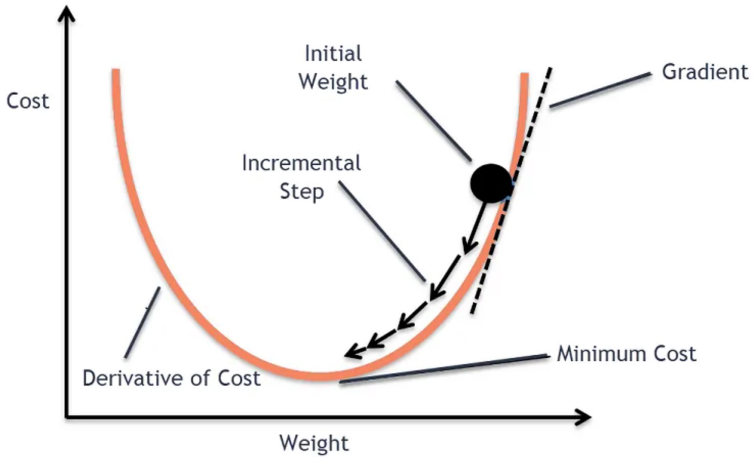
Loss acts as a feedback signal for the learning process:

- It tells the network how far off its predictions are from the desired outputs.
- Loss minimization is the ultimate goal during training.

Loss is minimized during training using **gradient descent**.

Gradient Descent

Gradient Descent is an optimization algorithm used to minimize the **loss** by iteratively adjusting the model's parameters (weights and biases). To find the optimal set of parameters that minimize the error between the predicted output $\hat{y}$ and the actual output y.



Steps in Gradient Descent

1. **Compute the Loss:** Use the current model parameters to calculate the loss for the training data.
2. **Calculate Gradients:** Determine how much the loss changes with respect to each parameter by computing the **gradient** (partial derivatives of the loss function).

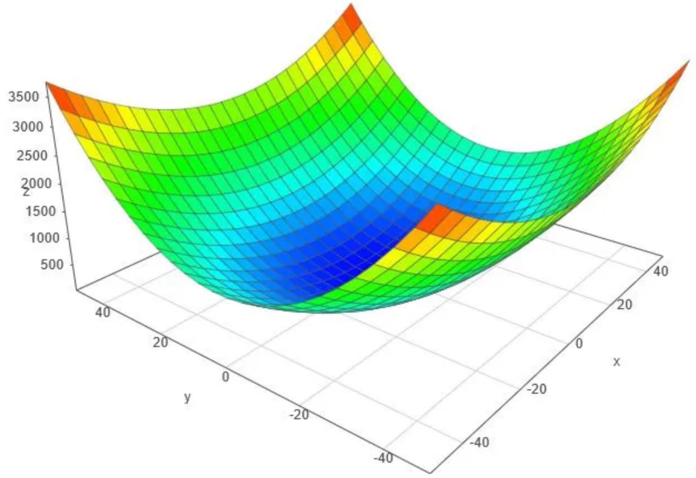
$$\frac{\partial \text{Loss}}{\partial w}, \frac{\partial \text{Loss}}{\partial b}$$

3. **Update Parameters:** Adjust the model parameters in the direction of the negative gradient (to reduce the loss):

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{\partial \text{Loss}}{\partial w}$$

$$b_{\text{new}} = b_{\text{old}} - \eta \cdot \frac{\partial \text{Loss}}{\partial b}$$

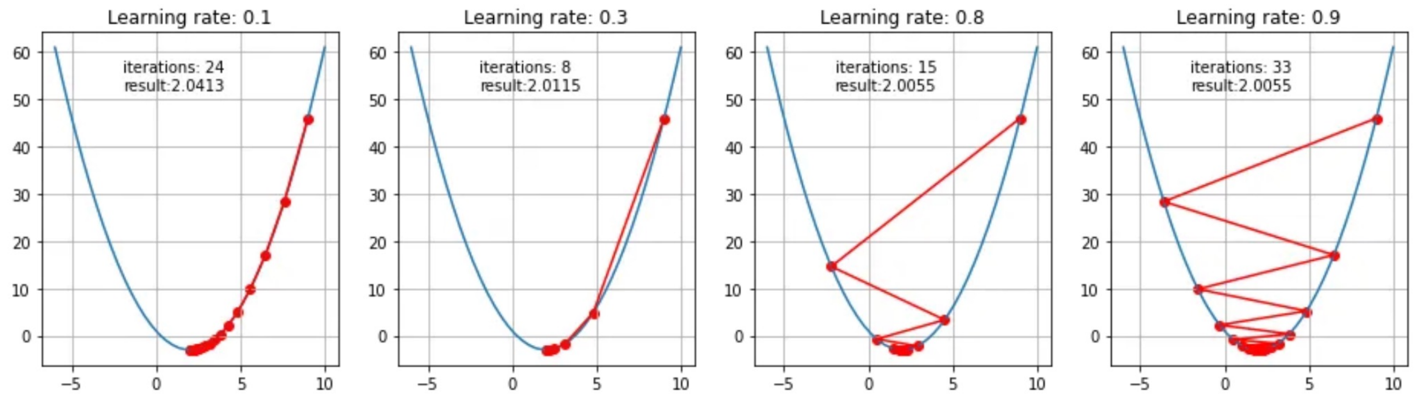
learning rate

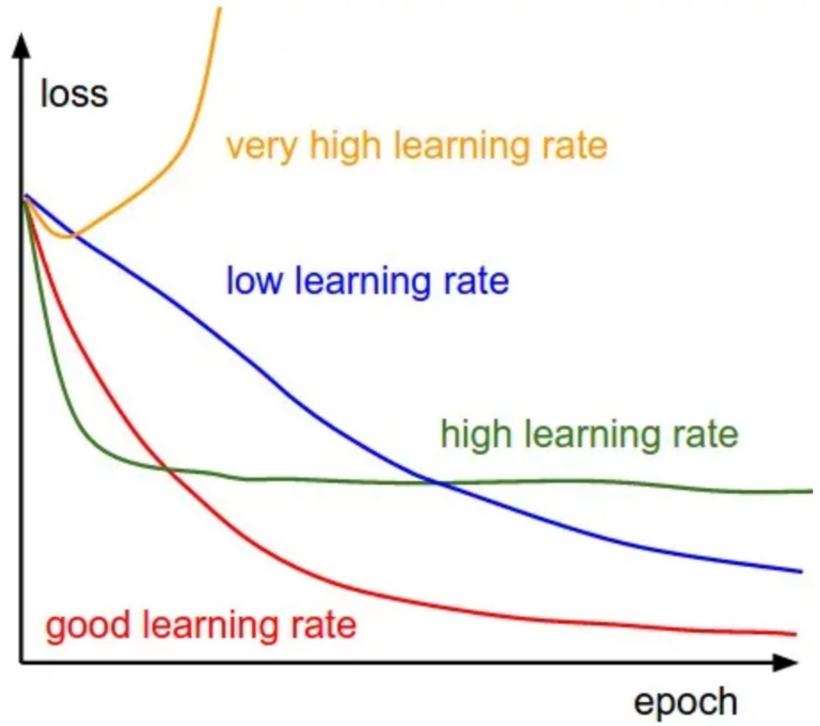


Learning Rate

One of the most important hyperparameter that determines the size of the steps taken during parameter updates.

- **Too Small:** Slow convergence, requiring many iterations.
- **Too Large:** May overshoot the minimum or diverge entirely.





https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21
https://www.analyticsvidhya.com/blog/2020/10/how-does-the-gradient-descent-algorithm-work-in-machine-learning/

Types of Gradient Descent

Batch Gradient Descent:

- Uses the entire dataset to compute the gradients in each iteration.
- **Pros:** Stable convergence.
- **Cons:** Computationally expensive for large datasets.

Stochastic Gradient Descent (SGD):

- Uses a single data point (randomly chosen) to compute the gradients.
- **Pros:** Faster updates, can escape local minima.
- **Cons:** Noisy updates, may lead to instability.

Mini-Batch Gradient Descent:

- Uses a small subset (mini-batch) of data to compute the gradients.

Backpropagation

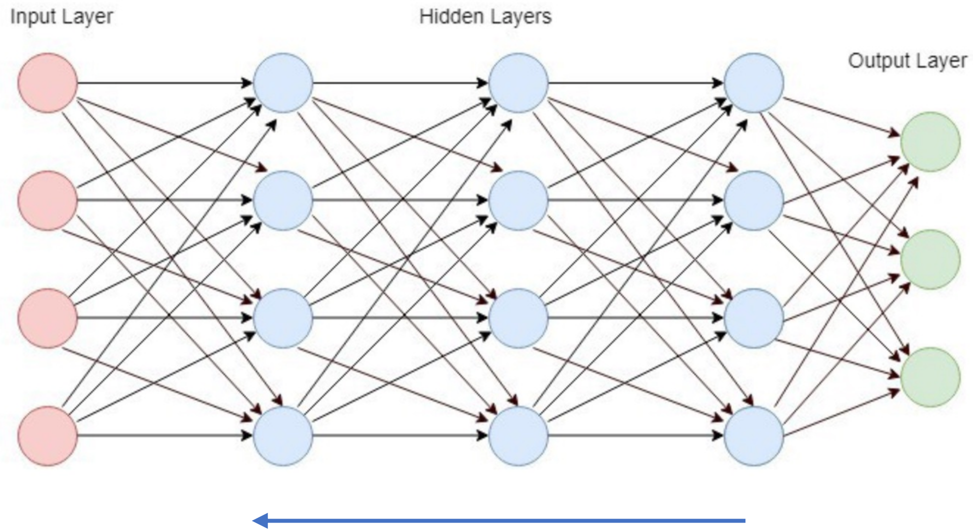
Propagate the error backward through the network to compute gradients of the loss with respect to each parameter.

Chain Rule. Neural networks are composed of layers of functions. The output of one layer becomes the input for the next, forming a chain of operations.

Backpropagation uses the chain rule of calculus to compute partial derivatives layer by layer:

$$\frac{\partial \text{Loss}}{\partial w^{(l)}} = \frac{\partial \text{Loss}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial w^{(l)}}$$

where l represents the layer, $z^{(l)}$ is the weighted sum, and $w^{(l)}$ is the weight.



Backpropagation

Challenges:

- Vanishing Gradients: Gradients become very small in deep networks, slowing learning. Common with sigmoid or tanh activations.
- Exploding Gradients: Gradients grow too large, destabilizing training. Common in recurrent networks.

How to work with text data

Challenges with text data

Unstructured Data

- Tweets, news, financial reports, earnings call transcript
- Irregularities: misspellings, abbreviations, and noise
- Test data is unstructured and varied

High Dimensionality

- Large vocabulary size leads to sparse representations
- The curse of dimensionality in processing words
- Increased computational inefficiency and memory usage

Semantic Ambiguity

- Same word can have multiple meanings based on context
- E.g., “bank” as in riverbank vs. financial institution

Order Dependency

- Word sequences affects meaning

Numerical input requirement

- Neural networks process numbers, not strings
- Text must be mapped to numerical representations

Early approaches to text representation

One-hot encoding

- Represents each word as a unique vector
- Sparse and high-dimensional
- Context-insensitive: distances among days of week are all the same

Case Analysis Economics and Finance

Case	[1 0 0 0 0]
Analysis	[0 1 0 0 0]
Economics	[0 0 1 0 0]
and	[0 0 0 1 0]
Finance	[0 0 0 0 1]

Bag-of-words

- Simple count-based representation
- Ignores word order and syntax

Two simple movie reviews:
Review A: "The movie was not good."
Review B: "The movie was good, not bad."

Word	The	Movie	Was	Not	Good	Bad
Review A	1	1	1	1	1	0
Review B	1	1	1	1	1	1

Early approaches to text representation

TF-IDF (Term Frequency-Inverse Document Frequency)

- Improves upon simple Bag-of-Words by not only considering word frequency but also accounting for how unique or rare a word is across the corpus
- Highlights unique, informative words in a document while downplaying common words (e.g., "the," "and")

- Straightforward to compute, scalable to moderately large corpora
- Words are treated as independent entities, ignoring semantic and syntactic relationships
- Produces sparse matrices for large vocabularies, which can lead to computational inefficiency
- Cannot handle words not seen in the training corpus

a. Term Frequency (TF):

- Measures how often a word appears in a document.
- Formula:

$$\mathrm{TF}(t, d) = \frac{\text{Frequency of term } t \text{ in document } d}{\text{Total number of terms in document } d}$$

b. Inverse Document Frequency (IDF):

- Measures how unique a word is across the corpus.
- Formula:

$$\mathrm{IDF}(t, D) = \log \frac{\text{Total number of documents in corpus } D}{1 + \text{Number of documents containing term } t}$$

  - Adding 1 to the denominator prevents division by zero for rare words.

c. TF-IDF Score:

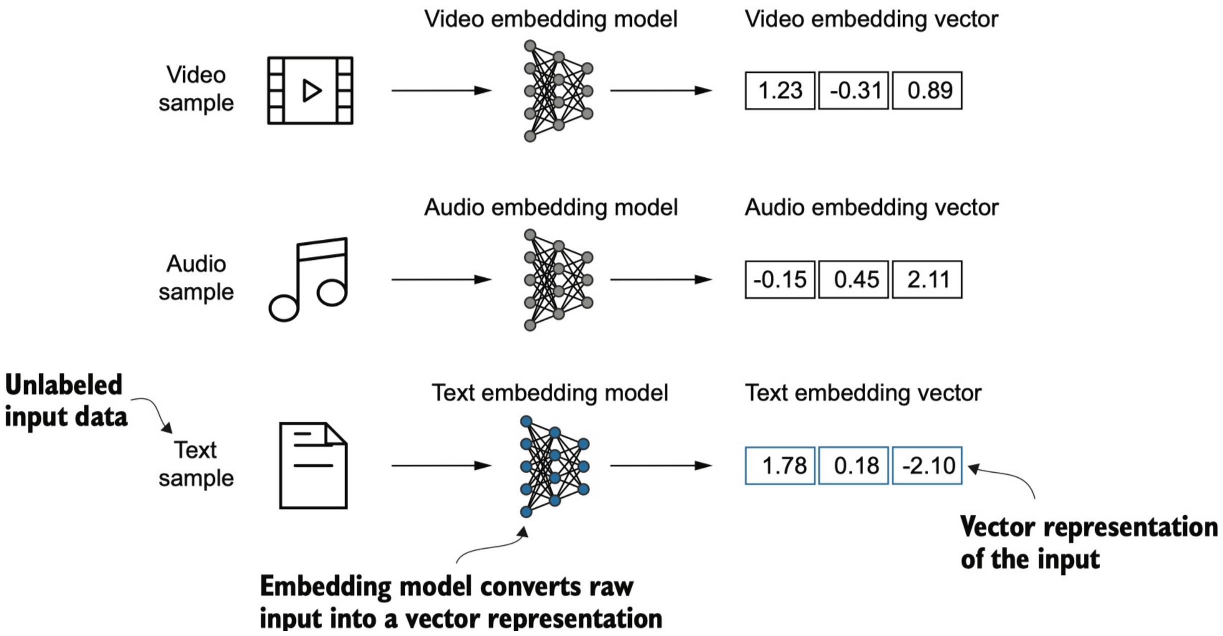
- Combines TF and IDF to give a score for each word in each document.
- Formula:

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

Word Embedding

A modern approach to text representation

Definition	Semantic understanding	Dimensionality reduction
• Dense vector representations of words in a continuous vector space • Encodes meaning, context, and relationships between words	• Captures sematic and syntactic information • Words with similar meanings are close in vector space	• Reduce sparsity • Embeddings map words into dense vectors (e.g., 300 dimensions instead of 100,000+ for one-hot)



Raschka, Sebastian. *Build a Large Language Model (From Scratch)*. Shelter Island, NY: Manning Publications, 2024.

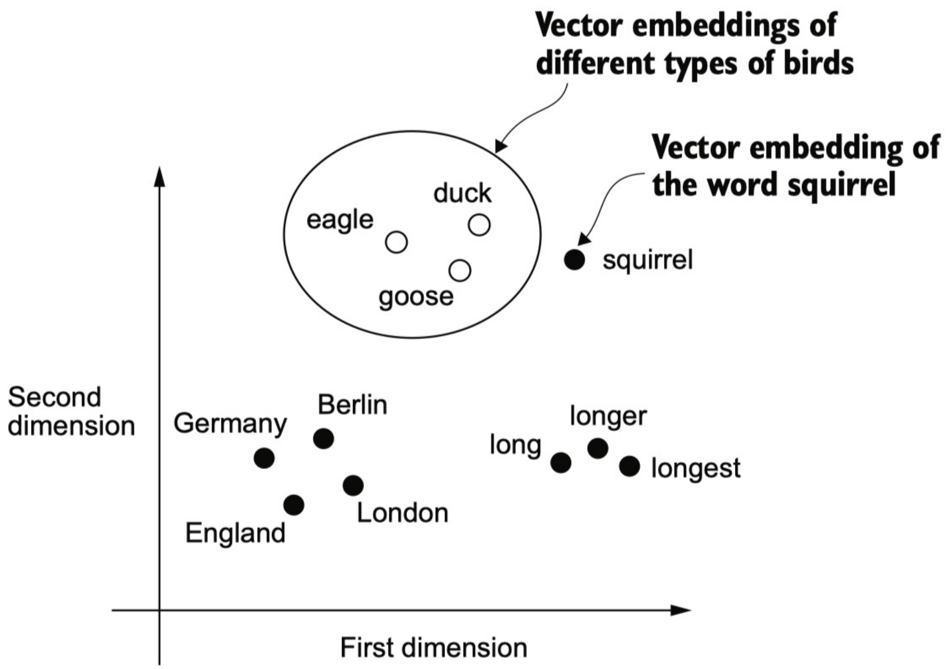
Visualizing word embeddings

Semantic clustering

- Words like “king”, “queen”, “prince”, and “princess” cluster together

Arithmetic operations

- king – man + woman = queen



Dimensionality reduction techniques like t-SNE and PCA map embeddings into 2D space for visualization

Raschka, Sebastian. *Build a Large Language Model (From Scratch)*. Shelter Island, NY: Manning Publications, 2024.

Evolvement of word embedding

Static to contextual representations

Static word embeddings

- Fixed vector for each word (independent of context)
- Examples: Word2Vec, Glove, FastText
- Transferable knowledge: reuse pretrained embeddings across tasks and domains

Contextual word embeddings

- Vectors change based on the context in which a word appears
- Example: BERT, GPT

Will be cover in later lecture

<u>Comparison</u>

- Static embeddings are faster to compute but limited in handling cases where the same word can have multiple meanings
- Contextual embeddings are more powerful but computationally intensive

How are word embeddings created

Popular techniques and models for static word embeddings

Word2Vec

- Continuous Bag of Words (CBOW): predicts a word based on surrounding context
- Skip-gram: predicts context words given a target word

GloVe (Global Vectors)

- Captures statistical co-occurrence of words in a corpus
- Focuses on global corpus statistics

FastText

- Embeds subword units (e.g., “kingdom” includes “king” and “dom”
- Handles out-of-vocabulary (OOV) words better

Word2Vec

Popular techniques and models for static word embeddings

Key idea:

- Introduced by Google in 2013
- Words appearing in similar contexts have similar meanings and are placed close together in the embedding space
- Captures both syntactic and semantic relationships

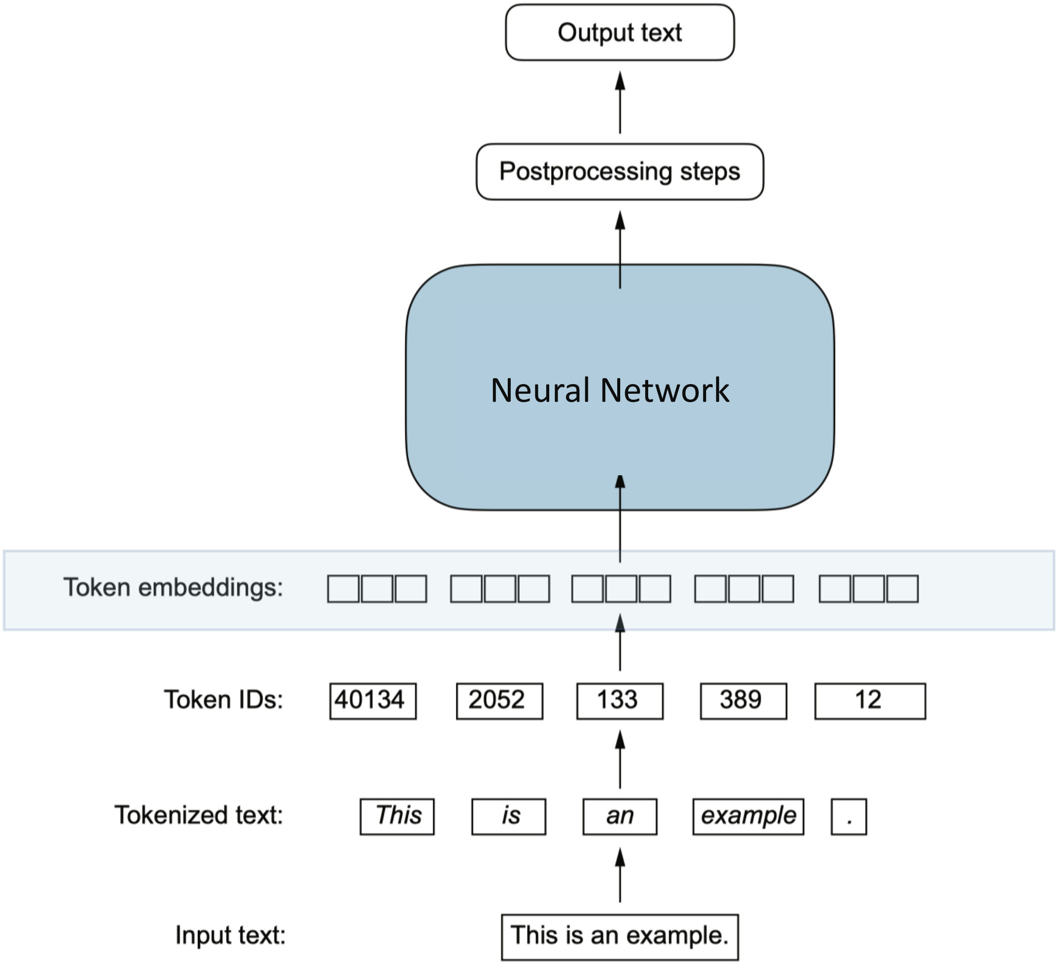
Continuous Bag of Words (CBOW)

- Predicts a target word based on its context words.
- Input: context words [w(i-2), w(i-1), w(i+1), w(i+2)]
- Output: target word w(i)

Skip-Gram

- Predicts context words given a target word.
- Input: target word w(i)
- Output: context words [w(i-2), w(i-1), w(i+1), w(i+2)]

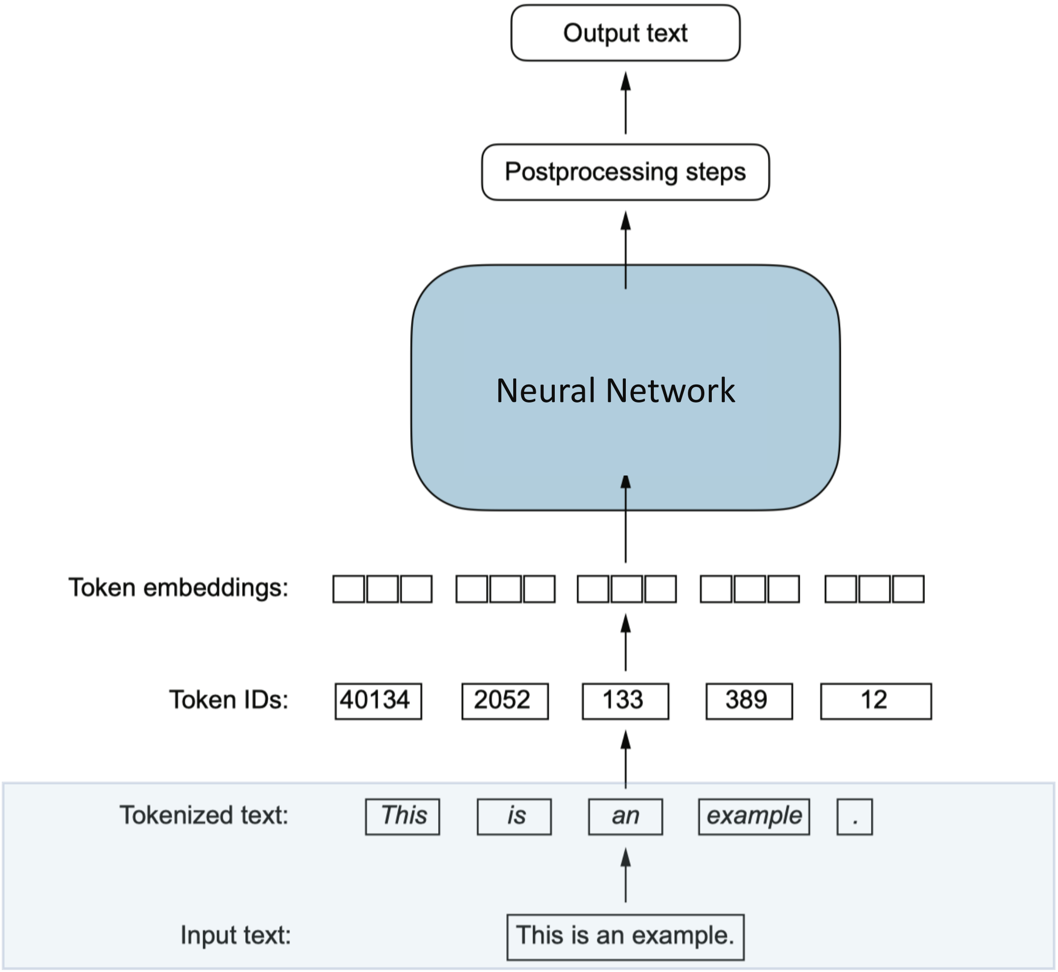
Word embedding



Raschka, Sebastian. *Build a Large Language Model (From Scratch)*. Shelter Island, NY: Manning Publications, 2024.

Tokenization

The process of breaking down text into smaller units called tokens (words, subwords, or characters)



Raschka, Sebastian. *Build a Large Language Model (From Scratch)*. Shelter Island, NY: Manning Publications, 2024.

Types of Tokenization

Word Tokenization

- Splits text into individual words.
- Typically using spaces as delimiters.
- Example: "The cat sat on the mat." → ["The", "cat", "sat", "on", "the", "mat"]

Pros:

- Easy to implement and understand.
- Matches human intuition (words are natural units of meaning).

Cons:

- Vocabulary size can become very large for diverse corpora.
- Fails to handle rare or out-of-vocabulary (OOV) words effectively.

Character Tokenization

- Splits text into individual characters, treating each as a token.
- Ignores word boundaries and punctuation significance.
- Example: "Cat" → ["C", "a", "t"]

Pros:

- Completely avoids OOV issues (all words are composed of characters).
- Smaller vocabulary (alphabet size + basic symbols).
- Language agnostic (works well for non-space-separated languages).

Cons:

- Loses semantic meaning of words (e.g., "play" vs. "pray" might appear similar at the character level).
- Longer sequences increase computational complexity.
- Limited applicability for tasks requiring semantic understanding.

Subword Tokenization

- Breaks words into smaller units like morphemes or frequent subword chunks.
- Uses techniques like Byte Pair Encoding (BPE), WordPiece, or Unigram.

Pros:

- Combines the strengths of word and character tokenization.
- Effectively handles rare and compound words.
- Reduces vocabulary size without sacrificing semantic information.

Cons:

- Computationally intensive during preprocessing and tokenization.
- Subword boundaries may not align with human intuition (e.g., splitting common words unnecessarily).
- Requires careful tuning (e.g., vocabulary size, merge operations).

Word Tokenization

```python
# Importing libraries for word tokenization
import nltk
from nltk.tokenize import word_tokenize

# Downloading tokenizer models
nltk.download('punkt')

# Example sentence
text = "Hello, world! This is an example of word tokenization."

# Perform word tokenization
tokens = word_tokenize(text)

# Display the tokens
print("Original Text:", text)
print("Tokens:", tokens)
```

```
Original Text: Hello, world! This is an example of word
tokenization.

Tokens: ['Hello', ',', 'world', '!', 'This', 'is', 'an',
'example', 'of', 'word', 'tokenization', '.']
```

```python
import re

# Example sentence
text = "Hello, world! This is an example of word tokenization."

# Perform word tokenization using regular expressions
tokens = re.findall(r'\b\w+\b', text)

# Display the tokens
print("Original Text:", text)
print("Tokens:", tokens)
```

```
Original Text: Hello, world! This is an example of word
tokenization.

Tokens: ['Hello', 'world', 'This', 'is', 'an', 'example', 'of',
'word', 'tokenization']
```

Explanation:

1. `\b` : Matches word boundaries.
2. `\w+` : Matches one or more word characters (letters, numbers, or underscores).
3. `re.findall()` : Finds all substrings that match the regular expression.

Character Tokenization

```python
# List of Pokémon names
pokemon_names = ["Pikachu", "Charmander", "Bulbasaur", "Squirtle", "Jigglypuff"]

# Function to tokenize a single name into characters
def tokenize_to_characters(name):
    return [char for char in name]

# Perform character tokenization for each Pokémon name
char_tokens = [tokenize_to_characters(name) for name in pokemon_names]

# Display results
for name, tokens in zip(pokemon_names, char_tokens):
    print(f"Original Name: {name} -> Character Tokens: {tokens}")
```

```
Original Name: Pikachu -> Character Tokens: ['P', 'i', 'k', 'a', 'c', 'h', 'u’]
Original Name: Charmander -> Character Tokens: ['C', 'h', 'a', 'r', 'm', 'a', 'n', 'd', 'e', 'r’]
Original Name: Bulbasaur -> Character Tokens: ['B', 'u', 'l', 'b', 'a', 's', 'a', 'u', 'r’]
Original Name: Squirtle -> Character Tokens: ['S', 'q', 'u', 'i', 'r', 't', 'l', 'e’]
Original Name: Jigglypuff -> Character Tokens: ['J', 'i', 'g', 'g', 'l', 'y', 'p', 'u', 'f', 'f']
```

Subword Tokenization – Byte-pair encoding (BPE)

Definition:

- Byte-Pair Encoding (BPE) is a subword tokenization technique that iteratively merges frequent pairs of characters or character sequences into subwords.
- Balances between character-level and word-level tokenization.

Key Features:

- Reduces vocabulary size compared to word-level tokenization.
- Handles out-of-vocabulary (OOV) words by splitting them into known subwords.
- Commonly used in modern NLP models (e.g., GPT, T5).

How BPE works

1. **Initialization:**
   - Start with a corpus where each word is split into characters and appended with a special end-of-word marker (e.g., w -> w_).
2. **Count Character Pair Frequencies:**
   - Count how often each adjacent pair of characters appears in the corpus.
3. **Merge Most Frequent Pair:**
   - Merge the most frequent pair into a single unit.
4. **Repeat Iteratively:**
   - Continue merging until a predefined number of merges is reached or vocabulary size is sufficient.
5. **Encoding Unknown Words:**
   - For unseen words, BPE splits them into known subwords.

"low low low low low lower lower newest newest newest newest newest newest widest widest widest"

low_: 5, lower_: 2, newest_: 6, widest_: 3

(l,o,w,_): 5, (l,o,w,e,r,_): 2, (n,e,w,e,s,t,_): 6, (w,i,d,e,s,t,_): 3

Merge 1: Merge the most frequent pair *(e, s),* which occurs *6 + 3 = 9* times, to form the newly merged symbol *'es'.*

(l,o,w,_): 5, (l,o,w,e,r,_): 2, (n,e,w,es,t,_): 6, (w,i,d,es,t,_): 3

Merge 2: Merge the most frequent pair *(es, t),* which occurs *6 + 3 = 9* times, to form the newly merged symbol *'est'.* Update the vocabulary and replace every occurrence of *(es, t)* with *'est'*:

(l,o,w,_): 5, (l,o,w,e,r,_): 2, (n,e,w,est,_): 6, (w,i,d,est,_): 3

Merge 3: Merge the most frequent pair *(est, _),* which occurs *6 + 3 = 9* times, to form the newly merged symbol *'est_'.* Update the vocabulary and replace every occurrence of *(est, _)* with *'est_'*:

(l,o,w,_): 5, (l,o,w,e,r,_): 2, (n,e,w,est_): 6, (w,i,d,est_): 3

Merge 4: Merge the most frequent pair *(l, o),* which occurs *5 + 2 = 7* times, to form the newly merged symbol *'lo'.* Update the vocabulary and replace every occurrence of *(l, o)* with *'lo'*:

(lo,w,_): 5, (lo,w,e,r,_): 2, (n,e,w,est_): 6, (w,i,d,est_): 3

Merge 5: Merge the most frequent pair *(lo, w),* which occurs *5 + 2 = 7* times, to form the newly merged symbol *'low'.* Update the vocabulary and replace every occurrence of *(lo, w)* with *'low'*:

(low,_): 5, (low,e,r,_): 2, (n,e,w,est_): 6, (w,i,d,est_): 3

Vocabulary	**vocabs = l, o, w, e, r, n, s, t, i, d, _, es, est, est_, lo, low**
Merge Rules	**(e, s) → es, (es, t) → est, (est, _) → est_, (l, o) → lo, (lo, w) → low**

How BPE works

1. **Initialization:**
   - Start with a corpus where each word is split into characters and appended with a special end-of-word marker (e.g., w -> w_).
2. **Count Character Pair Frequencies:**
   - Count how often each adjacent pair of characters appears in the corpus.
3. **Merge Most Frequent Pair:**
   - Merge the most frequent pair into a single unit.
4. **Repeat Iteratively:**
   - Continue merging until a predefined number of merges is reached or vocabulary size is sufficient.
5. **Encoding Unknown Words:**
   - For unseen words, BPE splits them into known subwords.

Tokenize new text: “newest binded lowers”

(newest_, binded_, lowers_)
(n, e, w, e, s, t, _), (b, i, n, d, e, d, _), (l, o, w, e, r, s, _)

Repetitively apply the merged rules in their learned order

(n, e, w, est_), (b, i, n, d, e, d, _), (low, e, r, s, _)

Any token not in the vocabulary will be replaced by an unknown token *“[UNK]”*:

(n, e, w, est_), ([UNK], i, n, d, e, d, _), (low, e, r, s, _)

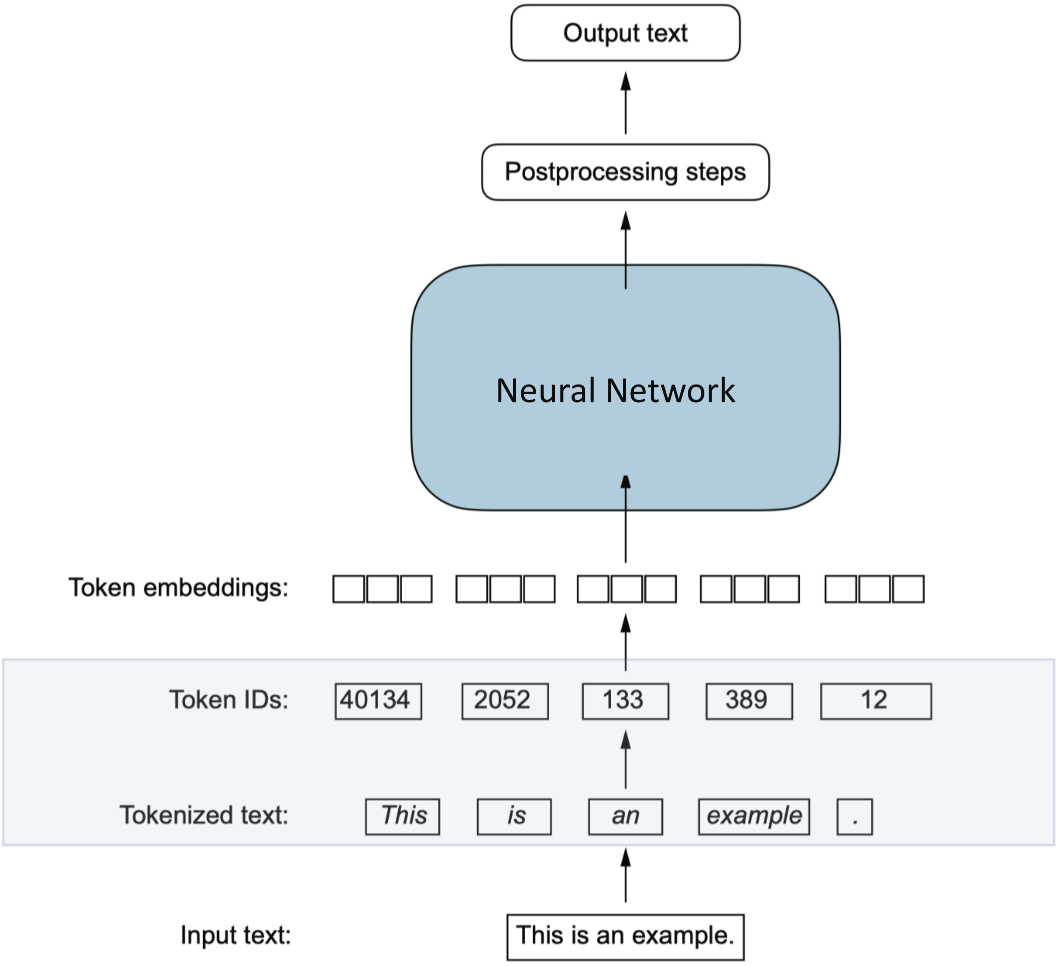
Result of Tokenization
[n, e, w, est_, [UNK], i, n, d, e, d, _, low, e, r, s, _]

Final Vocabulary	**vocabs = l, o, w, e, r, n, s, t, i, d, _, es, est, est_, lo, low**
Merge Rules	**(e, s) → es, (es, t) → est, (est, _) → est_, (l, o) → lo, (lo, w) → low**

https://medium.com/@hsinhungw/understanding-byte-pair-encoding-fd196ebfe93f

Tokenization

The process of breaking down text into smaller units called tokens (words, subwords, or characters)



Raschka, Sebastian. *Build a Large Language Model (From Scratch)*. Shelter Island, NY: Manning Publications, 2024.

Converting tokens into token IDs

- A token must be converted from a string to an integer to produce the token ID.
- Only numerical representations (token IDs) can be processed by neural networks
- Token IDs correspond to unique integers representing tokens in a vocabulary.

- **Vocabulary Construction:** Build a mapping between tokens and unique IDs.
- **Encoding:** Convert a sequence of tokens into a sequence of token IDs using the vocabulary.
- **Decoding:** Convert token IDs back into their corresponding tokens for interpretation or debugging.

```python
# Example tokens
tokens = ["hello", "world", "this", "is", "a", "test", "<unk>"]

# Step 1: Vocabulary construction
vocab = {token: idx for idx, token in enumerate(tokens)}
vocab["<unk>"] = len(vocab)  # Add an unknown token for OOV handling

# Step 2: Encode function (convert tokens to token IDs)
def encode(tokens, vocab):
    return [vocab.get(token, vocab["<unk>"]) for token in tokens]

# Step 3: Decode function (convert token IDs back to tokens)
def decode(token_ids, vocab):
    reverse_vocab = {idx: token for token, idx in vocab.items()}
    return [reverse_vocab.get(token_id, "<unk>") for token_id in token_ids]

# Example: Encoding and decoding
example_tokens = ["hello", "this", "is", "unknown_word"]
encoded_ids = encode(example_tokens, vocab)
decoded_tokens = decode(encoded_ids, vocab)
```

```
Tokens: ['hello', 'this', 'is', 'unknown_word']
Encoded Token IDs: [0, 2, 3, 6]
Decoded Tokens: ['hello', 'this', 'is', '<unk>']
```

Recurrent Neural Networks

The Importance of Sequential Data

Examples of Sequential Data

- Time-series (stock prices, temperature readings).
- Text (words in a sentence).
- Audio (sound wave over time).
- Sensor data (IoT readings).

Dependency Over Time

- Events at one time step can influence outcomes at future time steps.

Feedforward NN vs. RNN

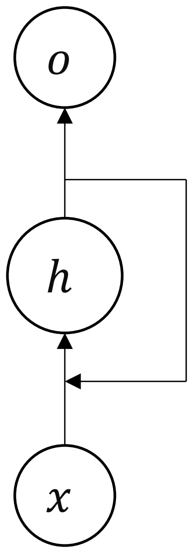
- Feedforward NNs ignore ordering.
- RNNs explicitly handle temporal or sequential dependency.

What is an RNN?

Neural network with recurrent (feedback) connections that store *hidden states*.

Memory of Past Inputs

- Tasks like predicting the next word in a sentence require information from previous words to make accurate predictions.
- Recurrent Neural Networks introduce a mechanism where the output from one step is fed back as input to the next, allowing them to retain information from previous inputs.
- The defining feature of RNNs is their hidden state h which preserves essential information from previous inputs in the sequence.



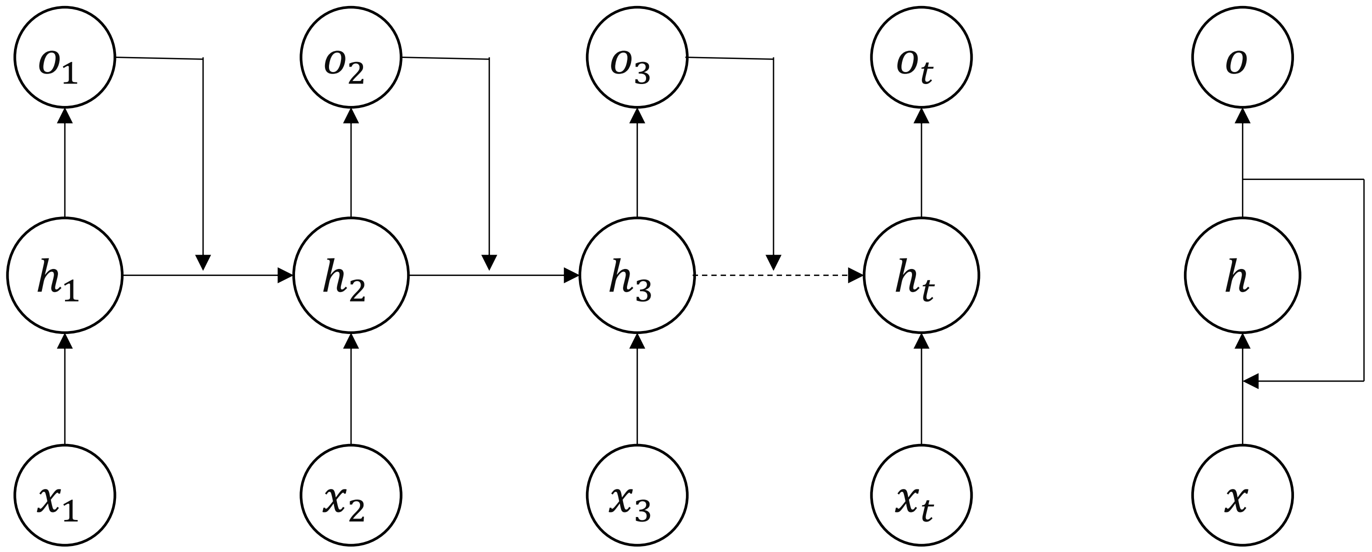
Can we do this with feedforward neural networks?

Memory of Past Inputs

- At each time step t, the network receives input x_t and a hidden state h_{t-1} from the previous step, produces a new hidden state h_t.

Parameter Sharing

- To reduce the model size and make sure the neural network perform consistently for the same inputs, we can force the the same weights to be applied at each time step



Architecture & Equations

Hidden State Update

$$h_t = f(W_x x_t + W_h h_{t-1} + b)$$

h_t: hidden state at time t

x_t: input at time t

W_x, W_h: learnable weight matrices

b: bias term

f: activation function (commonly $\tanh$ or ReLU)

Output Computation

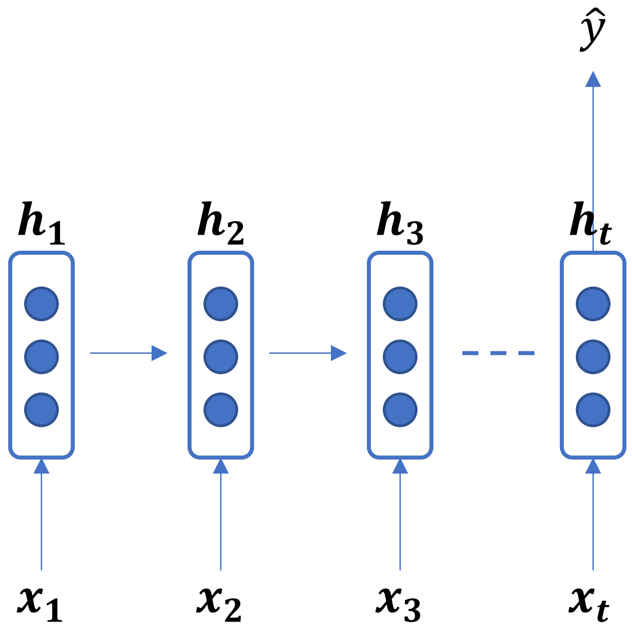
$$y_t = g(W_y h_t + c)$$

y_t: output at time t

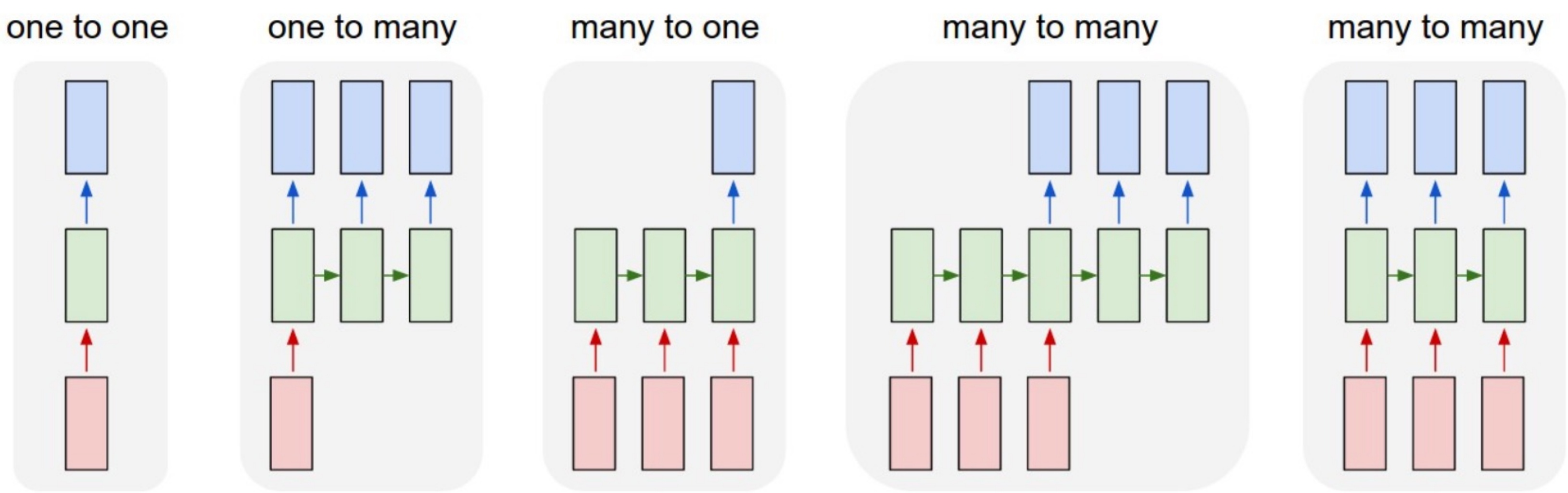
W_y, c: learnable output weight and bias

g: output activation (e.g., softmax for classification)

Types of Recurrent Neural Network



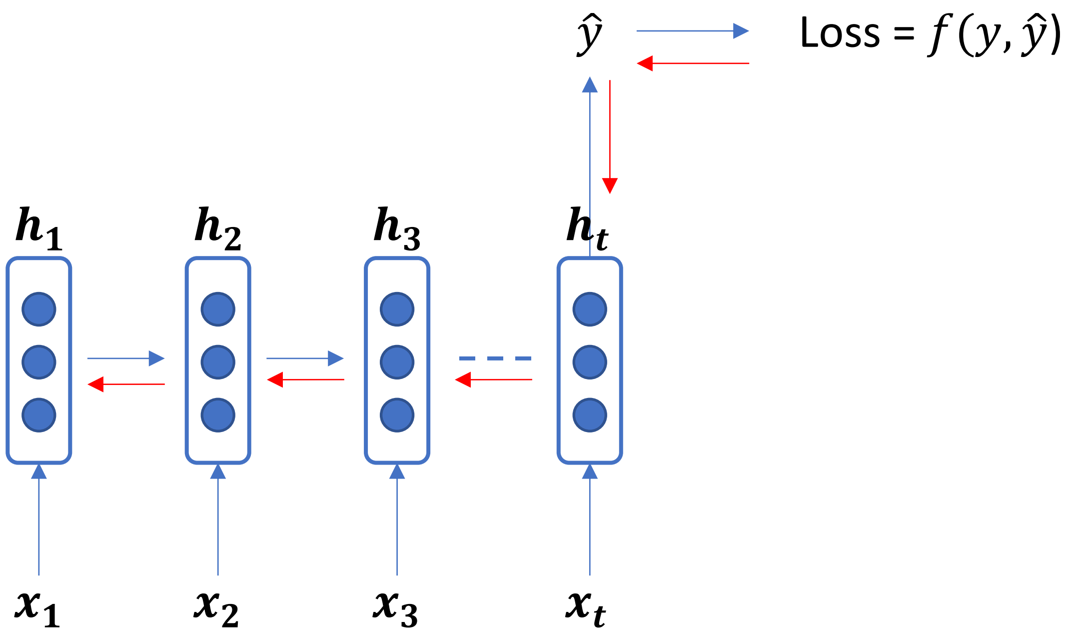
many to one example



Examples of sequence data in applications

Language Model	Speech Recognition	Machine Translation	Stock Prediction
Sequence to one	Sequence to sequence	Sequence to sequence	Sequence to one
X: text sequence Y: next word	X: wave sequence Y: text sequence	X: text sequence (in one language) Y: text sequence (in another language)	X: sequence of market data Y: next day/year price/direction

Backpropagation through time



Vanishing gradients and exploding gradient problem

Vanishing Gradients

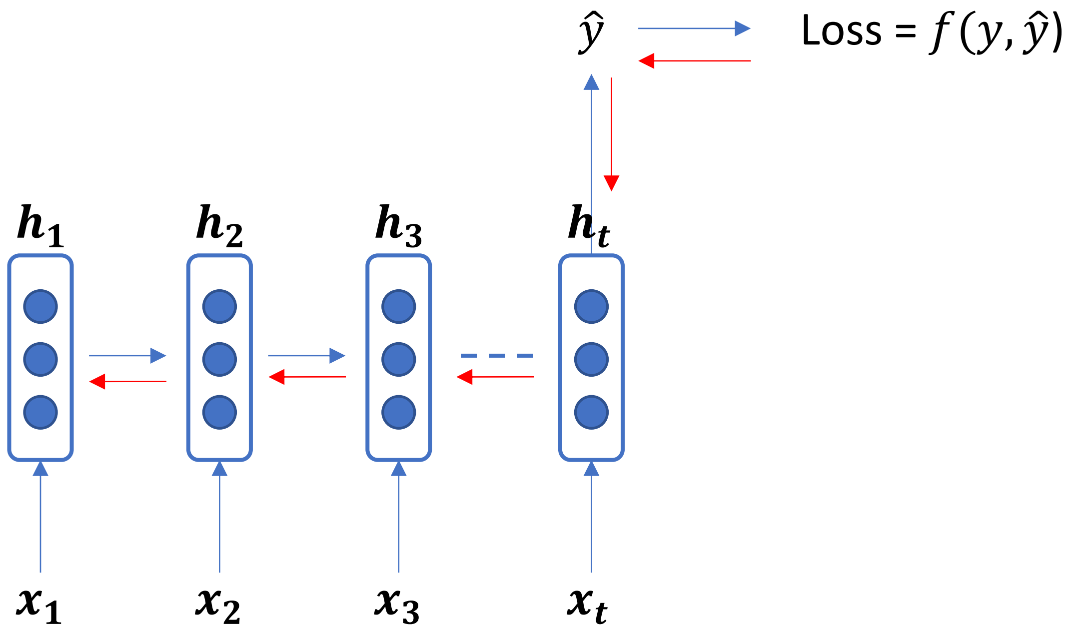
- Gradients decay exponentially through time steps, hindering learning of long-range dependencies.
- Signals from the past become too small to influence weight updates in earlier layers of the RNN.
- Difficult to learn long-range dependencies—the network struggles to connect distant historical information to the current output.

Exploding Gradients

- Gradients grow exponentially large during backpropagation
- Training becomes unstable or may diverge entirely.
- Model parameters can oscillate wildly instead of converging.

Why They Occur

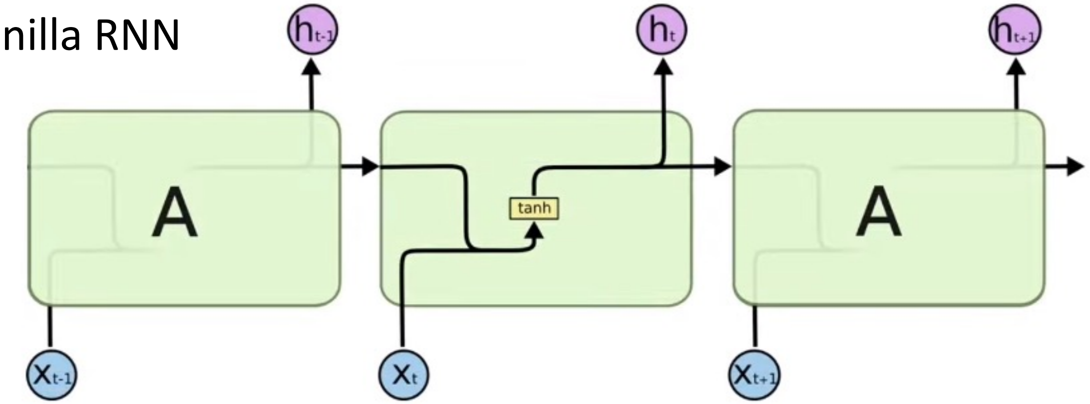
- Repeated multiplication of weight matrices over many time steps.
- In an RNN, each time step reuses the same weight matrices, compounding any factors > 1 or < 1.
- The more time steps you unroll, the greater the risk of these problems.



The chain rule: $\sigma'(h_t) \times \sigma'(h_{t-1}) \times \cdots \times \sigma'(h_1)$

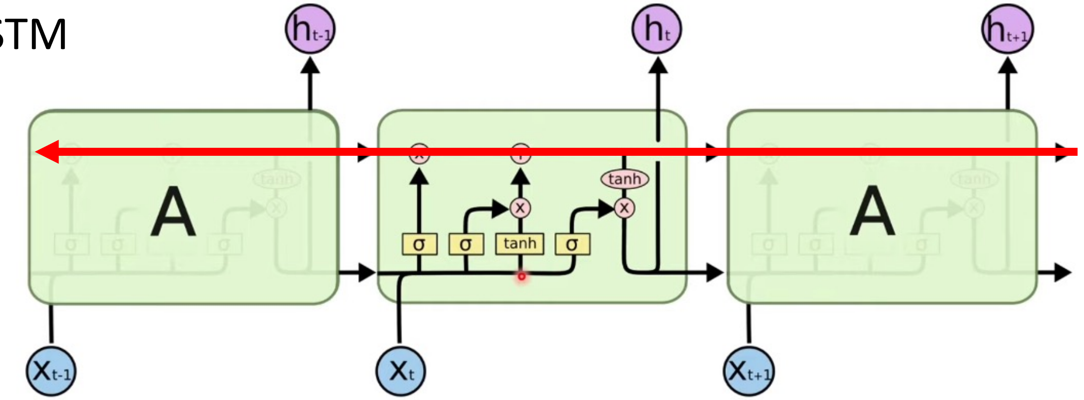
Long Short-Term Memory (LSTM) Networks

Vanilla RNN



- LSTM has gates to optionally let information through
- LSTM can decide how much old information to forget and how much new information to remember

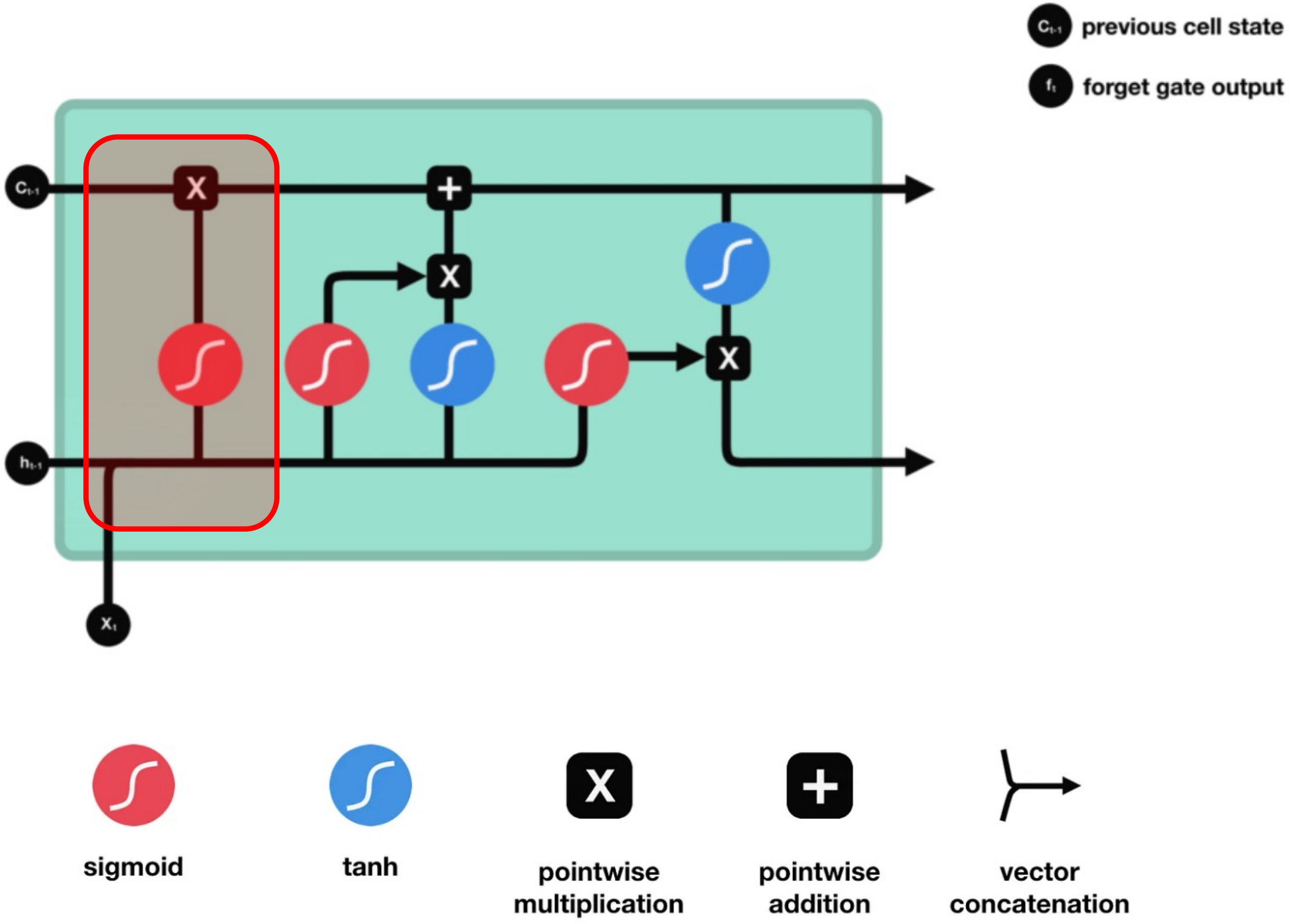
LSTM



- A highway for gradients to pass through
- Similar to ResNet for computer vision

LSTM Networks – Forget Gate

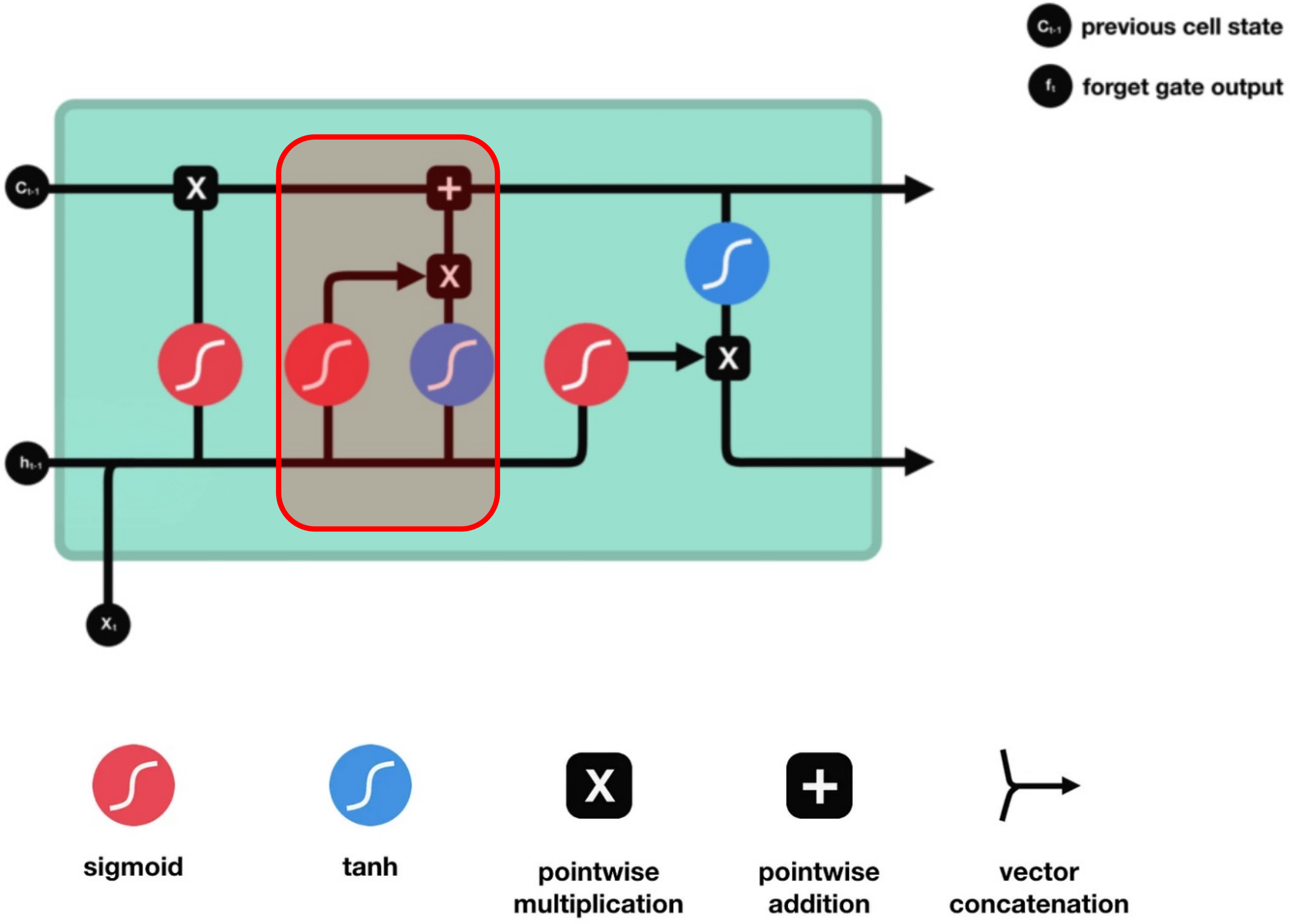
- Forget gate: how much information in previous cell state shall be kept or forgotten
- Input of sigmoid: previous hidden states and current input
- Output of sigmoid: value between 0 and 1
    - 0: forget all previous cell info
    - 1: keep all previous cell info



https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

LSTM Networks – Input Gate

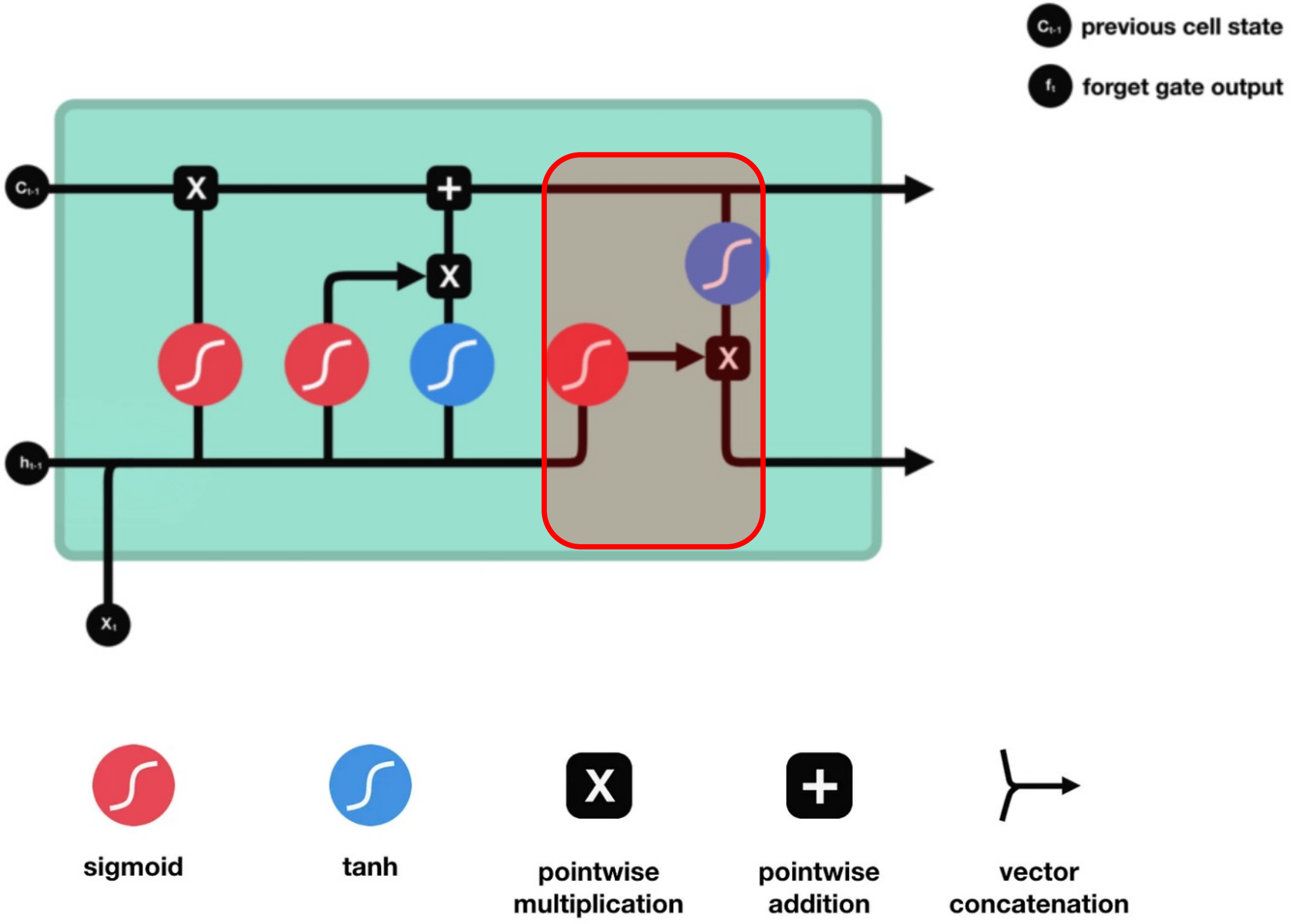
- Input gate: how much new information to be added to the cell state
- Input of sigmoid: previous hidden states and current input
- Output of sigmoid: value between 0 and 1 to decide which values are important
- Output of tanh: regulate the value to be between -1 and 1
- Multiply tanh output with sigmoid output: discount non-important information from the tanh output



https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

LSTM Networks – Output Gate

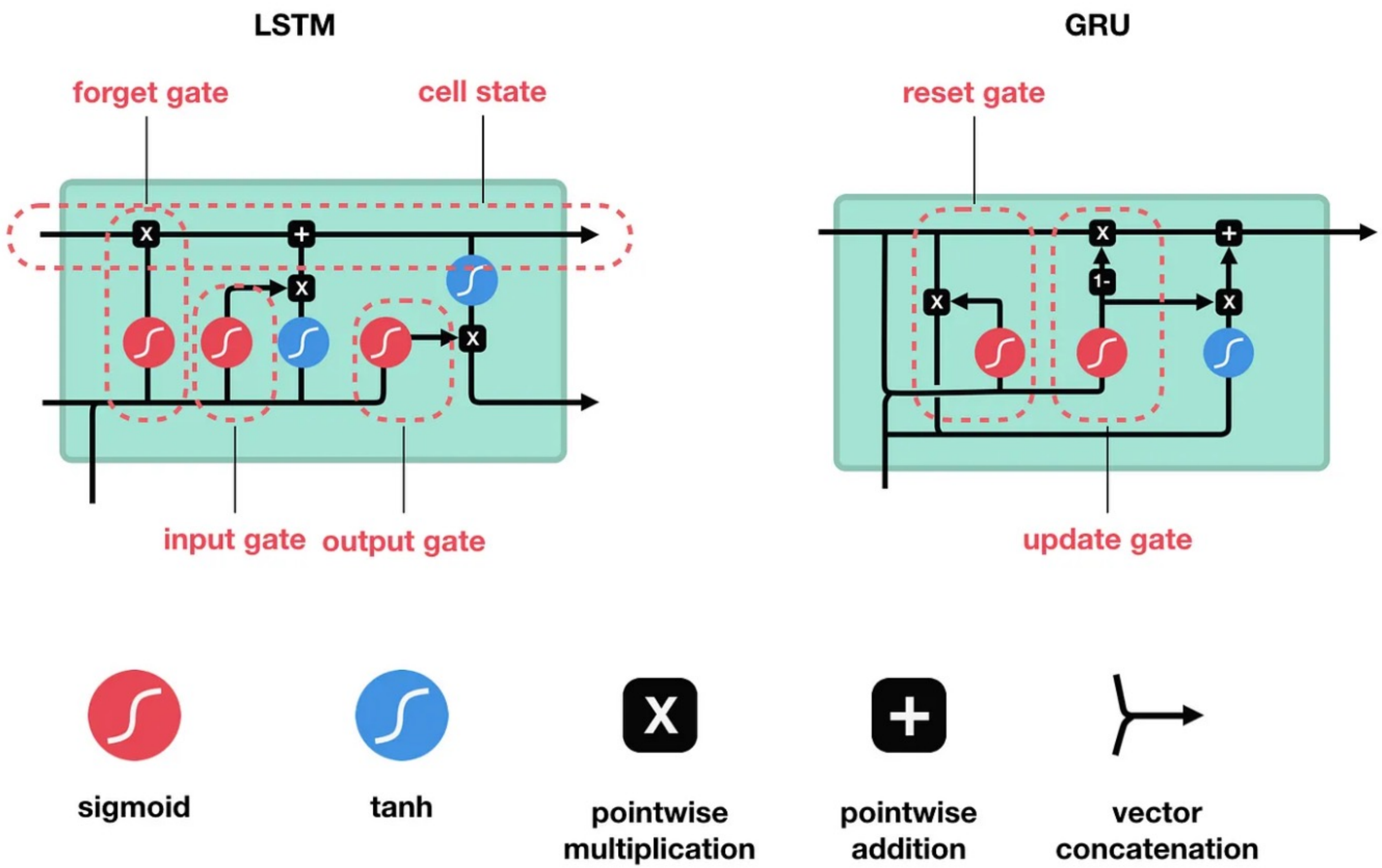
- Output gate: what shall be the next hidden state
- Input of sigmoid: previous hidden states and current input
- Output of sigmoid: value between 0 and 1 to decide which information hidden state shall carry forward
- Pass the newly updated cell state through a tanh function, then multiply with sigmoid output
- Result will be the updated hidden state



https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Gated Recurrent Units (GRUs)

- GRU is simpler than LSTM, and can be used to build much bigger networks
- LSTM is more general and powerful
- Both LSTM and GRU employs **Gating Mechanism** to address the issue of long term dependencies



https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Comparing LSTM, GRU, and Vanilla RNN

Vanilla RNN

Pros

- Simple architecture with fewer parameters.
- Easier to implement and faster per-step computation (fewer gates compared to LSTM/GRU).

Cons

- Highly susceptible to vanishing and exploding gradients, limiting its ability to capture long-range dependencies.
- Often underperforms on complex, lengthy sequences (e.g., long text documents, extended time-series).

Use Case Fit

- Short sequences or tasks where long-term memory is not critical.
- Educational or proof-of-concept scenarios (to illustrate the basics of recurrence).

Comparing LSTM, GRU, and Vanilla RNN

LSTM (Long Short-Term Memory)

Pros

- Designed to overcome vanishing gradients by using a cell state and gating mechanisms (forget, input, output gates).
- Excellent at capturing long-range dependencies, making it well-suited for tasks with extended context (e.g., full sentences, multi-week time-series).

Cons

- More complex structure → more parameters to learn, can be slightly slower to train compared to vanilla RNN or GRU.
- Potentially overkill for tasks with short sequences or straightforward patterns.

Use Case Fit

- Text-heavy tasks (language modeling, machine translation), long time-series forecasting, or any scenario needing robust memory of distant events.

Comparing LSTM, GRU, and Vanilla RNN

GRU (Gated Recurrent Unit)

Pros

- Simpler than LSTM—only two gates (reset and update)—leading to fewer parameters and often faster training.
- Frequently matches LSTM performance on many datasets, especially with moderate sequence lengths.

Cons

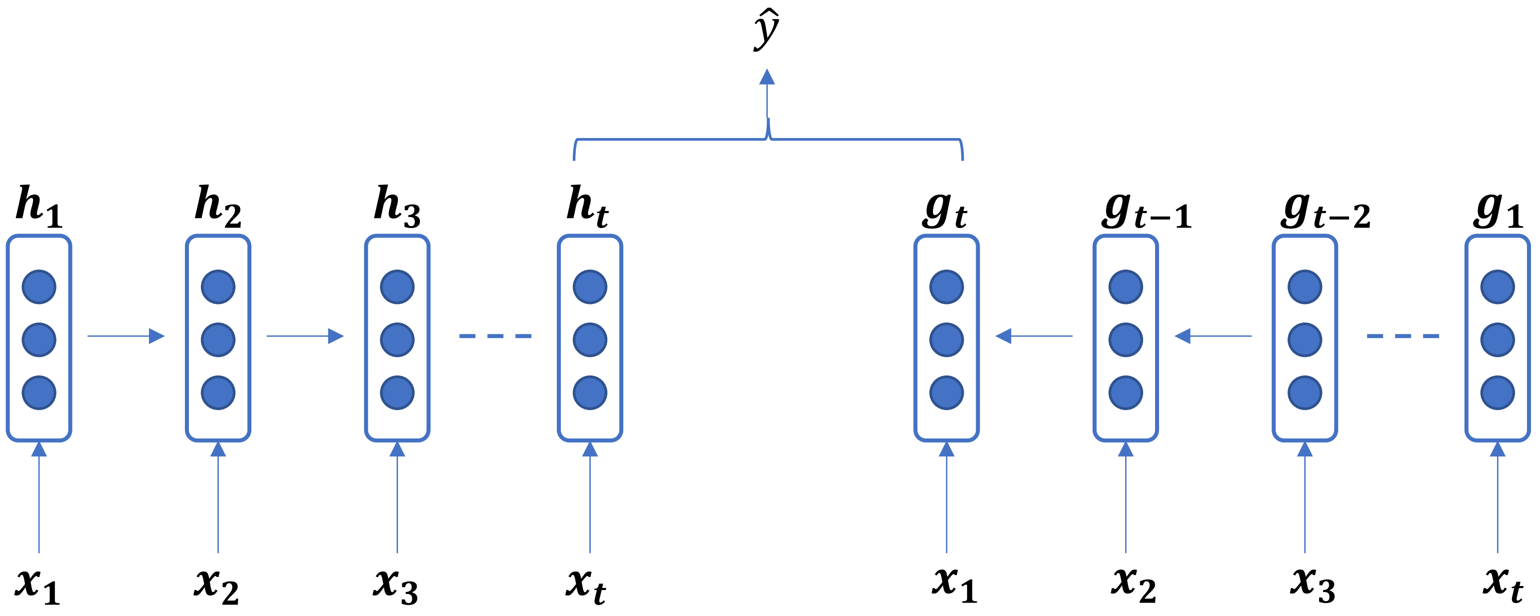
- May not capture extremely long dependencies *quite* as effectively as an LSTM in certain tasks.
- Slightly less interpretability regarding separate “memory cell” vs. hidden state (everything is combined into one).

Use Case Fit

- When you need faster training or have a smaller dataset.
- Highly popular for time-series, speech, and many NLP tasks where LSTM-like memory is desired but with less computational overhead.

Bidirectional Recurrent Neural Networks (Bi-RNNs)

- A *Bidirectional RNN* processes the input sequence *twice*: once from left to right (forward RNN) and once from right to left (backward RNN).
- The outputs from both directions are usually concatenated or otherwise combined at each time step.
- By looking *forward* and *backward* in time, the model can capture context from both the past and the future relative to each time step.



Why Use Bi-RNNs?

Enhanced Context

- In many tasks (e.g., text), knowing upcoming words is as important as knowing preceding words.
- Bi-RNNs help disambiguate contexts (for instance, a word might have multiple meanings that become clear only after reading the next few words).

Improved Accuracy

- Typically, Bi-RNNs yield better performance than unidirectional RNNs on tasks like speech recognition, text tagging, and machine translation where the future context is available.

Examples

- **NLP**: Part-of-speech tagging, Named Entity Recognition, where having both left and right context of a word improves tagging accuracy.
- **Speech**: Processing the entire audio clip allows the backward RNN to leverage information from later speech frames.