Neural Networks for NLP

RNN, LSTM, GRU & Named Entity Recognition

Neural Networks | Recurrent Models | NER

Learning Objectives

Neural Network Fundamentals for NLP

Feedforward networks, activation functions, backpropagation, word embeddings

Recurrent Neural Networks (RNN)

Sequential data, hidden states, vanishing gradient, applications

Deep Dive into LSTM Architecture

Cell state, forget/input/output gates, long-term dependencies

GRU as LSTM Alternative

Simplified gating, update/reset gates, LSTM comparison

Neural Networks to Named Entity Recognition

BIO tagging, BiLSTM-CRF, entity extraction

AGENDA

Session Overview

Neural Networks Fundamentals
Perceptron, MLP, Activation Functions

Recurrent Neural Networks (RNN)
Sequential Processing, Vanishing Gradients

Long Short-Term Memory (LSTM)
Gates, Cell State, Long Dependencies

Gated Recurrent Unit (GRU)
Simplified Architecture, Comparison

Named Entity Recognition (NER)
BIO Tagging, BiLSTM-CRF

Lab & Final Assignment Q&A
Sentiment Analysis with RNN/LSTM

REVIEW

Session 06 Recap

N-gram Language Models

- $P(w_n|w_1 \ldots w_{n-1}) \approx P(w_n|w_{n-k} \ldots w_{n-1})$
- Markov assumption for tractability
- Smoothing: Add-k, Kneser-Ney

Perplexity

- $PP = P(W)^{-1/N} = 2^{H(W)}$
- Lower perplexity = Better model

Word2Vec

- CBOW: Context → Center word
- Skip-gram: Center → Context words
- Negative Sampling for efficiency

GloVe

- Global co-occurrence matrix
- Count-based + Prediction hybrid

Key Takeaway

Word embeddings capture semantic relationships in dense vectors. Today we use these as input to neural networks!

Neural Networks

Fundamentals for NLP

From Perceptrons to Deep Learning

Perceptron • Multi-Layer Networks • Activation Functions • Backpropagation

Why Neural Networks for NLP?

Traditional ML Limitations

- Manual feature engineering required
- Bag-of-words loses word order
- N-grams have data sparsity
- Limited generalization ability

Neural Network Advantages

- Automatic feature learning
- Word embeddings preserve semantics
- Handle variable-length sequences
- Learn hierarchical representations

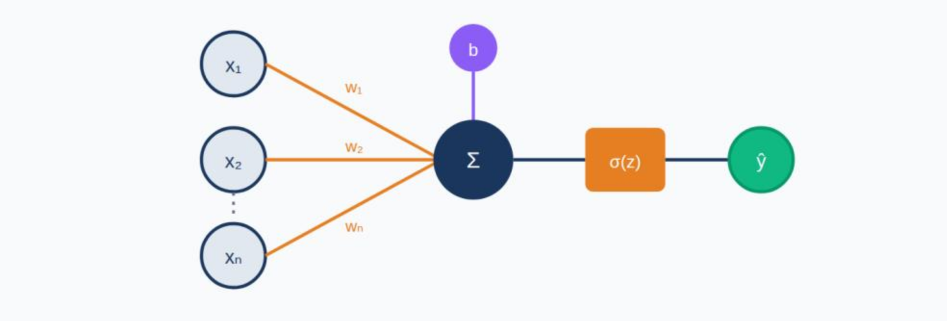
Evolution of NLP Models

Rule-based	Hand-crafted rules
Statistical	N-grams, HMM, CRF
Neural	Word2Vec, RNN, LSTM
Transformer	BERT, GPT, LLMs

Today's Focus: RNN, LSTM, GRU - foundation for sequence modeling before Transformers.

The Perceptron: Building Block

Perceptron Architecture



Mathematical Formula

$$z = \Sigma_i w_i x_i + b$$

$$\hat{y} = \sigma(z)$$

Components:

- **x:** Input features (word embeddings)
- **w:** Learnable weights
- **b:** Bias term
- **σ:** Activation function
- **ŷ:** Output prediction

NLP Application

Input x can be word embedding vectors (300-dim). Output for binary sentiment: positive/negative.

Activation Functions

Sigmoid

$\sigma(z) = 1 / (1 + e\text{-}z)$

- Output: (0, 1)
- Good for probabilities
- Vanishing gradient

Tanh

$\tanh(z) = (ez - e\text{-}z) / (ez + e\text{-}z)$

- Output: (-1, 1)
- Zero-centered
- Common in RNN/LSTM

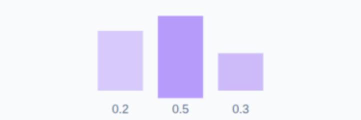
ReLU

$\text{ReLU}(z) = \max(0, z)$

- Output: [0, ∞)
- Fast computation
- Dying ReLU

Softmax

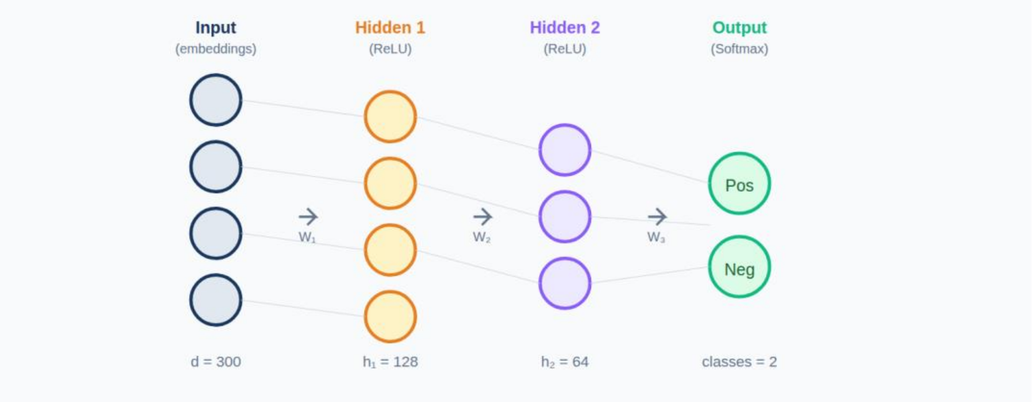
$\text{softmax}(z_i) = e^{z_i} / \Sigma_j e^{z_j}$



- Output: probability dist.
- Sum = 1
- Multi-class output

Multi-Layer Perceptron (MLP)

MLP Architecture for Text Classification



Forward Pass

$$h_1 = \mathrm{ReLU}(W_1 x + b_1)$$
$$h_2 = \mathrm{ReLU}(W_2 h_1 + b_2)$$
$$\hat{y} = \mathrm{Softmax}(W_3 h_2 + b_3)$$

📊 **Parameters**

- W_1: 300 × 128 = 38,400
- W_2: 128 × 64 = 8,192
- W_3: 64 × 2 = 128
- **Total: ~47K params**

MLP Limitation

MLP treats input as **fixed-size vector**. Can't handle **variable-length sequences** or **word order**!

Training: Backpropagation

Training Loop

Forward Pass
Compute $\hat{y} = f(x; W)$

Compute Loss
$L = \text{CrossEntropy}(y, \hat{y})$

Backward Pass
Compute $\partial L/\partial W$ (chain rule)

Update Weights
$W \leftarrow W - \eta \cdot \partial L/\partial W$

Chain Rule

$$\partial L/\partial W_1 = \partial L/\partial \hat{y} \cdot \partial \hat{y}/\partial h_2 \cdot \partial h_2/\partial h_1 \cdot \partial h_1/\partial W_1$$

Loss Functions

Cross-Entropy (Classification)

$$L = -\Sigma_i \, y_i \log(\hat{y}_i)$$

Binary Cross-Entropy

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

Optimizers

SGD: $W \mathrel{-}= \eta\nabla L$

Adam: Adaptive lr

RMSprop: Momentum

Insight

Backpropagation uses **chain rule** to compute gradients efficiently. In RNNs, this extends to **Backpropagation Through Time (BPTT)**.

Neural Networks for Text

Text → Neural Input

"The movie was great!"
Raw Text

↓

["the", "movie", "was", "great"]
Tokenization

↓

[42, 156, 89, 2301]
Token IDs

↓

[[0.2, -0.1, ...], [0.5, 0.3, ...], ...]
Word Embeddings (d=300)

Approach 1: Average Pooling

$$x\_avg = (1/n)\ \Sigma_i\ e_i$$

Simple but **loses word order**. "Dog bites man" = "Man bites dog"

Approach 2: Concatenation

$$x = [e_1;\ e_2;\ \ldots;\ e_n]$$

Preserves order but **fixed length** required. Padding needed!

Approach 3: Recurrent (RNN)

$$h_t = f(h_{t-1}, x_t)$$

Variable length, preserves order, captures **sequential patterns**!

Recurrent Neural Networks

Processing Sequential Data

Memory for Sequences

Architecture • Hidden State • BPTT • Vanishing Gradient

RNN: The Key Idea

Core Concept: Hidden State as Memory

$$h_t = f(h_{t-1}, x_t)$$

Current state = f(Previous state, Current input)

h_{t-1}
Memory from past

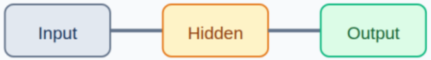
x_t
Current input

h_t
New memory

Why "Recurrent"?

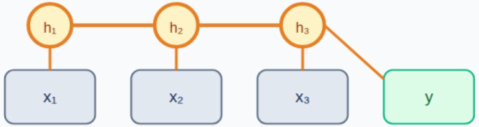
The **same function f** is applied at every time step. Output feeds back as input → creates a **loop/recurrence**.

Feedforward (MLP)



One-shot: No memory between inputs

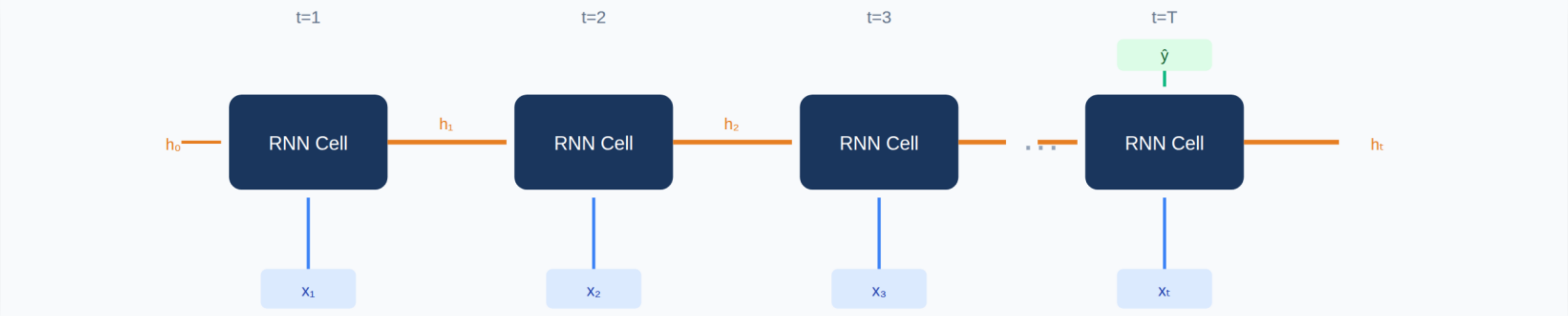
Recurrent (RNN)



Sequential: h carries information across time

Analogy: Reading a sentence - you remember previous words while reading the current one!

RNN Architecture: Unrolled View



RNN Equations

$$h_t = \tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b_h)$$

$$y_t = W_{hy} \cdot h_t + b_y$$

Shared Parameters

- **W_{xh}**: Input → Hidden weights
- **W_{hh}**: Hidden → Hidden weights
- **W_{hy}**: Hidden → Output weights
- Same W used at every time step!

Key Point

Weight sharing allows processing sequences of **any length** with fixed parameters!

RNN Architectures for NLP Tasks

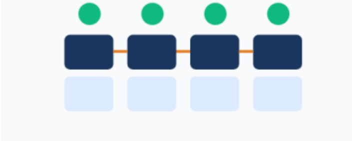
Many-to-One

Use: Sentiment Analysis, Classification

One-to-Many

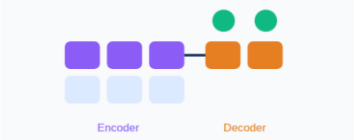
Use: Image Captioning, Text Gen

Many-to-Many (Same)



Use: POS Tagging, NER

Seq2Seq (Encoder-Decoder)



Use: MT, Summarization, QA

The Vanishing Gradient Problem

The Problem

During backpropagation, gradients are **multiplied** at each time step. For long sequences, gradients become **exponentially small** (vanish) or **exponentially large** (explode).

Mathematical View

$$\partial L/\partial W = \Sigma_t\ \partial L/\partial h_t \cdot \partial h_t/\partial h_{t-1} \cdot \ldots \cdot \partial h_1/\partial W$$

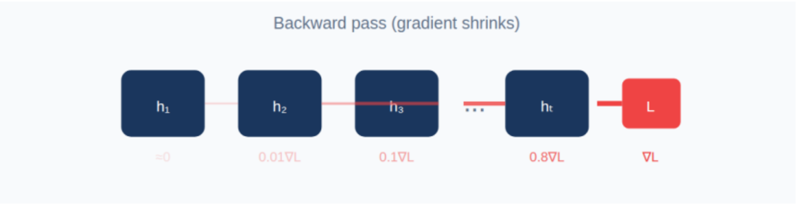
If $|\partial h_t/\partial h_{t-1}| < 1$
Gradient → 0
(Vanishing)

If $|\partial h_t/\partial h_{t-1}| > 1$
Gradient → ∞
(Exploding)

Quick Fix: Gradient Clipping

if ||g|| > threshold: g ← g × (threshold / ||g||)

Gradient Flow Visualization



Solutions

- **LSTM/GRU:** Gating mechanisms
- **Gradient Clipping:** Cap gradient magnitude
- **Skip Connections:** Residual connections
- **Better Initialization:** Xavier/He init

Real Impact: "The cat, which was sitting on the mat, _____" - RNN forgets about "cat" for long dependencies!

Long Short-Term Memory

LSTM Networks

Solving Long-Range Dependencies

Cell State • Forget Gate • Input Gate • Output Gate

LSTM: The Key Innovation

Two Types of Memory

Cell State (C_t)

Long-term memory "highway". Information flows with minimal change.

Hidden State (h_t)

Short-term/working memory. Filtered version of cell state.

Why LSTM Works

- **Cell state:** Direct path for gradients (no vanishing!)
- **Gates:** Control what to remember/forget
- **Additive updates:** C_t = forget + new (not multiplication)

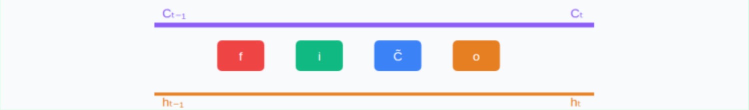
History: Introduced by Hochreiter & Schmidhuber (1997). Still widely used today for sequence modeling!

RNN vs LSTM Comparison

Simple RNN

Only 1 state: $h_t = \tanh(W \cdot [h_{t-1}, x_t])$

LSTM

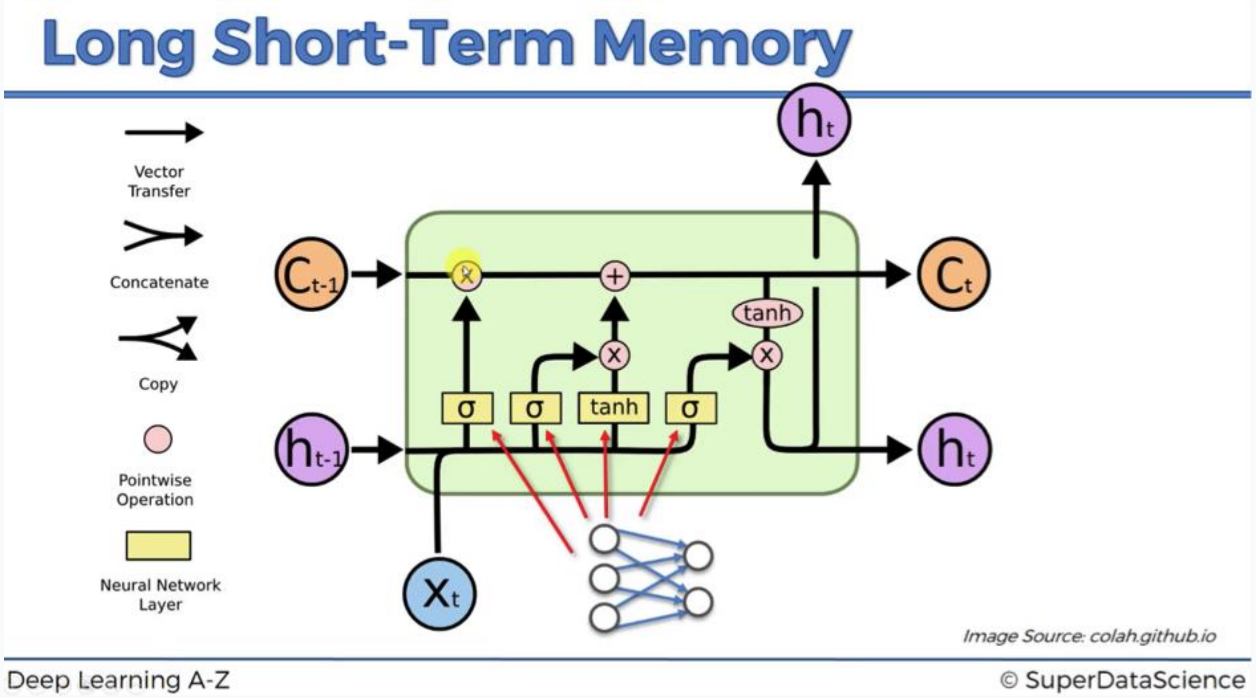


2 states + 3 gates: Forget, Input, Output

Key Insight

LSTM adds a **"highway"** for information (cell state) with **gates** to control traffic. Gradients flow easily!

LSTM: Architecture



Gate 1: Forget Gate

Purpose

Decide what information to DISCARD from cell state. Acts like a filter removing irrelevant past info.

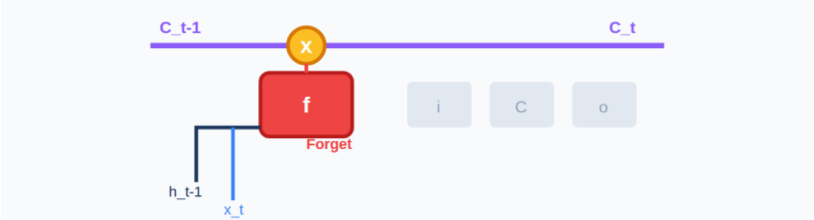
Formula

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

- f_t in (0, 1): Forget gate output
- σ : Sigmoid function (0 to 1)
- $[h_{t-1}, x_t]$: Concat hidden + input

Output: $f_t = 0$ means forget, $f_t = 1$ means keep

Forget Gate in LSTM



Example: Language Model

"The cat is sleeping. It purrs." When processing "It": Keep "cat" info, Forget "sleeping" details.

Gate 2: Input Gate + Candidate

Input Gate Purpose

Decide **what new information to STORE** in the cell state. Controls the flow of new data.

Input Gate Formula

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

Output $i_t \in (0,1)$: How much of new candidate to add

Candidate Cell State

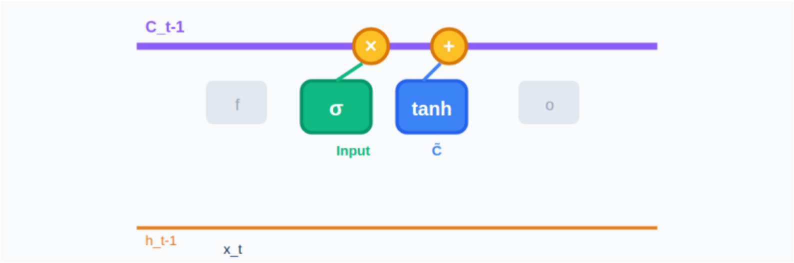
Create **new candidate values** that could be added to the cell state.

Candidate Formula

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Output $\tilde{C}_t \in (-1,1)$: Potential new values for cell

Input Gate + Candidate in LSTM



Combined Effect

$$\text{New info} = i_t \times \tilde{C}_t$$

Input gate filters which parts of candidate to actually add to cell state.

Example: Reading "The cat sat on the **mat**"

- Candidate $\tilde{C}$: encodes "mat" semantics
- Input gate: high value → store location info

Cell State Update

The Core LSTM Update

Combine forgetting old information and adding new information into a single update step.

Cell State Update Formula

$$C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t$$

Forget Term

$f_t \times C_{t-1}$

What to keep from past

Input Term

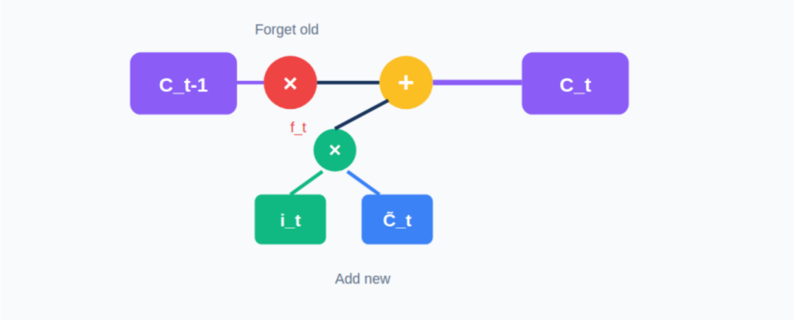
$i_t \times \tilde{C}_t$

What to add from now

Why This Works

ADDITION (not multiplication) enables gradients to flow directly through time. This is the key to solving vanishing gradients!

Cell State Update Flow



Gradient Highway

During backpropagation, gradients can flow directly through the **addition** operation. The gradient of C w.r.t. C includes an **additive path** that doesn't vanish!

Gate 3: Output Gate

Purpose

Decide what to OUTPUT from cell state. Filters cell to produce hidden state.

Output Gate Formula

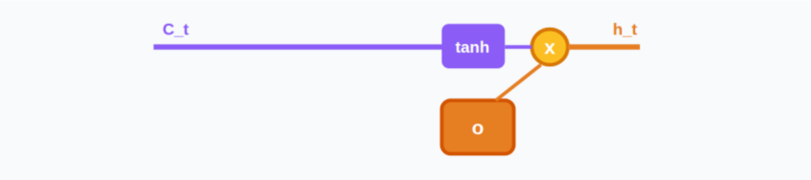
$$o_t = \text{sigma}(W_o \,.\, [h_{t-1}, x_t] + b_o)$$

Hidden State Formula

$$h\_t = o_t * \tanh(C_t)$$

tanh squashes to (-1,1), o_t filters output

Output Gate in LSTM



Two LSTM Outputs

C_t (Cell)
Internal memory

h_t (Hidden)
Visible output

Complete LSTM Equations

1. Forget Gate

$f_t = \text{sigma}(W_f \, . \, [h_{t-1}, x_t] + b_f)$

2. Input Gate

$i_t = \text{sigma}(W_i \, . \, [h_{t-1}, x_t] + b_i)$

3. Candidate Cell

$C_t = \tanh(W_C \, . \, [h_{t-1}, x_t] + b_C)$

4. Cell State Update

$C_t = f_t * C_{t-1} + i_t * C_t$

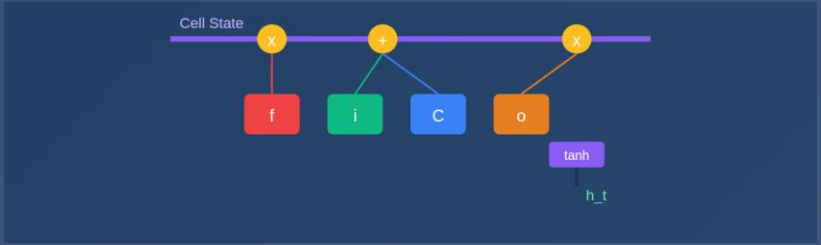
5. Output Gate

$o_t = \text{sigma}(W_o \, . \, [h_{t-1}, x_t] + b_o)$

6. Hidden State

$h_t = o_t * \tanh(C_t)$

LSTM Architecture



Parameter Count

$4 * ((n + m) * m + m)$ parameters

n = input dim, m = hidden dim

PART 4

GRU Architecture

Gated Recurrent Unit

A Simpler Alternative to LSTM

2 Gates • ~25% Fewer Params • Cho et al. 2014

GRU: Simplified Gating

Key Simplification

GRU merges cell state and hidden state into **one state**. Only **2 gates** instead of 3!

Update Gate (z_t)

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

Controls how much of the **previous state** to keep vs. **new candidate**. Combines forget + input gates!

Reset Gate (r_t)

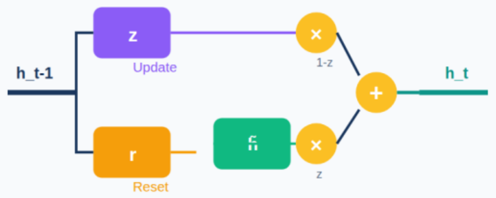
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

Controls how much **previous state** to use when computing new candidate. Can "reset" to ignore history.

Candidate State

$$\tilde{h}_t = \tanh(W \cdot [rt \times h_{t-1}, x_t] + b)$$

GRU Architecture



Final State Update

$$h_t = (1 - z_t) \times h_{t-1} + z_t \times \tilde{h}_t$$

z=0: keep old state | z=1: use new candidate

Insight: The update gate creates a "soft switch" between keeping the old state and using the new candidate. Linear interpolation!

LSTM vs GRU

LSTM (1997)

Pros

Better for very long sequences, explicit memory control, output gate flexibility

Cons

More parameters, slower training, more complex

GRU (2014)

Pros

~25% fewer params, faster training, simpler implementation

Cons

May struggle with very long dependencies

Rule of Thumb: Start with GRU for faster iteration. Switch to LSTM if you need longer memory.

PART 5

Named Entity Recognition

NER: Finding Real-World Entities

BiLSTM + CRF for Sequence Labeling

👤 Person 🏢 Organization 📍 Location 📅 Date

What is Named Entity Recognition?

Definition

NER is a **sequence labeling task** that identifies and classifies named entities (people, organizations, locations, etc.) in text.

Example

Apple ORG announced that Tim Cook PER will visit Tokyo LOC on Monday DATE

Key Challenges

Ambiguity: "Apple" = company or fruit? | **Multi-word:** "New York City" | **Unknown entities:** New names | **Context dependency:** Same word, different types

Common Entity Types

- 👤 **PER** — Person names
- 🏢 **ORG** — Organizations
- 📍 **LOC** — Locations
- 📅 **DATE** — Dates & times
- 💰 **MONEY** — Monetary values
- 🔣 **MISC** — Miscellaneous

BIO Tagging Scheme

What is BIO?

BIO labels each token with its position relative to an entity span.

Tag Meanings

- B - Begin entity
- I - Inside entity
- O - Outside entity

Schemes

- BIO: B-PER, I-PER, B-ORG...
- BIOES: +End, +Single

English: "Tim Cook works at Apple in Cupertino"

Token	Tag	Explanation
Tim	B-PER	Begin Person
Cook	I-PER	Inside Person
works	O	Outside
at	O	Outside
Apple	B-ORG	Begin Organization
in	O	Outside
Cupertino	B-LOC	Begin Location

Vietnamese: "Chủ tịch Nguyễn Văn A thăm Hà Nội"

Token	Tag	Explanation
Chủ_tịch	O	Outside (title)
Nguyễn	B-PER	Begin Person
Văn	I-PER	Inside Person
A	I-PER	Inside Person
thăm	O	Outside
Hà_Nội	B-LOC	Begin Location

NER Benchmark Datasets

CoNLL-2003 (English)

Most widely used benchmark

Entity Types: PER, LOC, ORG, MISC

Train: 14,987 sentences

Dev: 3,466 sentences

Test: 3,684 sentences

Source: Reuters news articles

```
# CoNLL format example
John    B-PER
lives   O
in      O
New     B-LOC
York    I-LOC
```

OntoNotes 5.0

Larger, more diverse

18 Entity Types:

PERSON, ORG, GPE, DATE, TIME, MONEY, PERCENT, CARDINAL, ...

Size: ~77K sentences

Sources: News, broadcast, web

OntoNotes vs CoNLL

More entity types (18 vs 4)

Multi-domain coverage

Harder benchmark

Vietnamese NER (VLSP)

Vietnamese Language and Speech Processing

Entity Types: PER, LOC, ORG

VLSP 2016: ~16K sentences

VLSP 2018: ~20K sentences

Challenges: Word segmentation

Dataset Selection

Research: CoNLL-2003 (standard)

Production: Domain-specific data

Vietnamese: VLSP datasets

Character-level Features for NER

Why Character Features?

- Handle OOV (out-of-vocabulary)
- Capture morphology (prefixes, suffixes)
- Learn capitalization patterns
- "John" vs "john" matters!

CharCNN

Convolve over character embeddings

"John" --> [J,o,h,n] --> CNN --> pool --> char_emb

Fast, captures n-gram patterns

CharLSTM

BiLSTM over character sequence

"John" --> BiLSTM(J,o,h,n) --> [h_fwd; h_bwd]

Better context, more parameters

```
# CharCNN Implementation
class CharCNN(nn.Module):

def __init__(self, char_vocab, char_emb_dim, out_dim):
  self.char_emb = nn.Embedding(char_vocab, char_emb_dim)
  self.conv = nn.Conv1d(char_emb_dim, out_dim,
  kernel_size=3, padding=1)

def forward(self, chars):
  x = self.char_emb(chars)  # (batch, word_len, emb)
  x = x.transpose(1, 2)     # (batch, emb, word_len)
  x = F.relu(self.conv(x))
  x = F.max_pool1d(x, x.size(2))  # Max over time
  return x.squeeze(2)
```

Combining with Word Embeddings

word_repr = [word_emb ; char_emb]

Concatenate before feeding to BiLSTM

Subword (BPE)

"playing" --> ["play", "##ing"]

Used in BERT, handles rare words

Comparison

CharCNN: Fast, local patterns

CharLSTM: Better, slower

Subword: BERT default

Recommendation

BiLSTM-CRF: Use CharCNN

Transformer: Subword (built-in)

BiLSTM for NER

Why Bidirectional?

NER benefits from **both past and future context**. A word's entity type often depends on words that come after it!

Example: Context Matters

"**Washington** crossed the river"
→ Forward: could be location or person
→ With future context: person (action verb follows)

"I flew to **Washington**"
→ With past context: location (travel verb before)

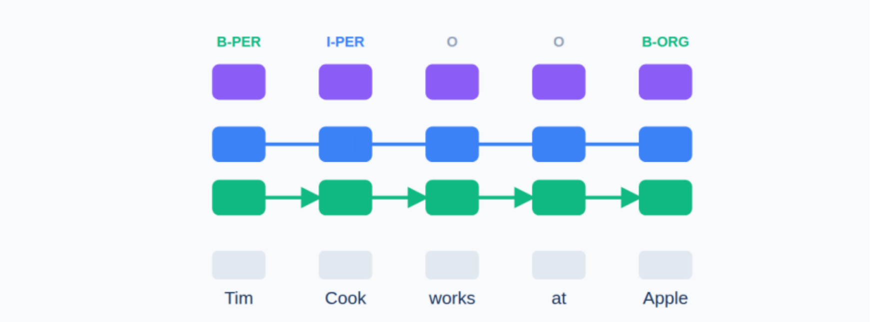
BiLSTM Architecture

→ **Forward LSTM**

← **Backward LSTM**

Concatenate: $[\overrightarrow{h}; \overleftarrow{h}]$

BiLSTM-NER Architecture



Output Computation

$$h_t = [\overrightarrow{h}_t ; \overleftarrow{h}_t] \rightarrow \text{softmax} \rightarrow \text{tag}$$

Concatenate forward & backward states, then classify each token.

Why Do We Need CRF Layer?

The Problem with Independent Tag Predictions

❌ Problem: Softmax Alone

BiLSTM + Softmax predicts each tag independently:

- Position 1: $P(tag|word_1) \rightarrow$ argmax
- Position 2: $P(tag|word_2) \rightarrow$ argmax
- Position 3: $P(tag|word_3) \rightarrow$ argmax

No consideration of **tag dependencies!**

✓ Solution: Add CRF Layer

CRF models the **joint probability** of the entire tag sequence:

$$P(y_1, y_2, \ldots, y_n \mid x)$$

Key components:

- **Emission scores** from BiLSTM
- **Transition scores** between tags (learned!)

Example Invalid Output:

"New York City" → **I-LOC I-LOC I-LOC**
I-LOC cannot START an entity! Must be B-LOC first.

Example Valid Output:

"New York City" → **B-LOC I-LOC I-LOC**
CRF learns: $T(O \rightarrow I\text{-}LOC) = -\infty$, forces B-LOC to start!

CRF Score: $Score(x, y) = \Sigma_i [\ Emission(y_i, x_i) + Transition(y_{i-1}, y_i)\ ]$ → Then find argmax using **Viterbi**

CRF Layer: Mathematical Details

CRF Score Function

Score of a label sequence y given input x:

$$s(x, y) = \text{Sum}_i [\, E(y_i) + T(y_{i-1}, y_i) \,]$$

$E(y_i)$: Emission score from BiLSTM (word y_i)

$T(y_{i-1}, y_i)$: Transition score (learnable matrix)

Partition Function Z(x)

Normalizing constant over ALL possible sequences:

$$Z(x) = \text{Sum}_{y'} \exp(s(x, y'))$$

Naive: exponential in sequence length!

Solution: Forward algorithm O(n * k^2)

Probability of Sequence

$$P(y|x) = \exp(s(x,y)) \,/\, Z(x)$$

Softmax over all possible label sequences

Forward Algorithm

Compute Z(x) efficiently via dynamic programming:

$$\text{alpha}_t(j) = \text{Sum}_i [\, \text{alpha}_{\{t-1\}}(i) * \exp(T_{ij} + E_j) \,]$$

$\text{alpha}_t(j)$: Sum of scores of all paths ending at tag j at position t

Final: $Z(x) = \text{Sum}_j \, \text{alpha}_n(j)$

Viterbi Decoding (Inference)

Find highest-scoring sequence:

$$y^* = \text{argmax}_y \, s(x, y)$$

Like Forward but with MAX instead of SUM

Backtrack to find best path

Why CRF Helps NER?

Transition matrix learns: I-PER cannot follow B-LOC

Ensures valid BIO sequences!

CRF Transition Matrix Visualization

Transition Matrix $T[i,j] = \text{Score}(tag_i \rightarrow tag_j)$

From \ To	O	B-PER	I-PER	B-LOC	I-LOC
O	0.5	0.8	-∞	0.7	-∞
B-PER	0.3	0.2	1.2	0.1	-∞
I-PER	0.4	0.3	0.9	0.2	-∞
B-LOC	0.4	0.3	-∞	0.2	1.1
I-LOC	0.5	0.4	-∞	0.3	0.8

Valid transition

-∞ = Invalid (blocked)

Invalid Transition Rules

Rule 1: I-X cannot start a sequence

O → I-PER = -∞

Rule 2: I-X must follow B-X or I-X (same type)

B-PER → I-LOC = -∞

Rule 3: I-X cannot follow different entity type

I-PER → I-LOC = -∞

Valid Sequence Examples

"John works at Google"

B-PER → O → O → B-ORG ✓

"New York City"

B-LOC → I-LOC → I-LOC ✓

Why -∞?

exp(-∞) = 0 → Invalid sequences have probability 0

CRF layer ENFORCES valid BIO tag constraints!

Viterbi Decoding: Step-by-Step Example

Input: "John lives in Paris" | Tags: O, B-PER, B-LOC

Viterbi Trellis (scores at each position)

Tag \ Word	John	lives	in	Paris
O	1.2	**3.2**	**3.8**	3.5
B-PER	**2.5**	2.1	1.9	2.0
B-LOC	0.8	1.5	2.2	**4.5**

Best Path: B-PER → O → O → B-LOC

B-PER → O → O → B-LOC

John lives in Paris

Viterbi Algorithm Steps

1. **Initialize: $\delta_1(\text{tag}) = \text{Emission}(\text{tag}, \text{word}_1)$**

   $\delta_1(\text{B-PER}) = E(\text{B-PER}, \text{John}) = 2.5$

2. **Recursion: For t = 2 to n**

   $\delta_t(j) = \max[\delta_{t-1}(i) + T(i,j) + E(j, \text{word}_t)]$

Example: $\delta_2(O)$ at 'lives'

From B-PER: 2.5 + T(B-PER,O) + E(O,lives)
= 2.5 + 0.5 + 0.2 = **3.2** ✓
From O: 1.2 + 0.5 + 0.2 = 1.9

3. **Backtrack: Follow pointers from max**

   Result: B-PER → O → O → B-LOC

Time Complexity: $O(n \times k^2)$ | n=words, k=tags

Why CRF Layer? Side-by-Side Comparison

Input sentence: "New York City is amazing"

❌ BiLSTM Alone (WRONG)

New	York	City	is	amazing
I-LOC	I-LOC	I-LOC	O	O

INVALID! I-LOC cannot start entity

Problems:

- **Softmax predicts each tag** independently
- No constraint between adjacent tags
- Entity must start with B-X, not I-X
- Results in invalid BIO sequences

↓ ↓ ↓ ↓ ↓

(Independent predictions - no sequence modeling)

✓ BiLSTM-CRF (CORRECT)

New	York	City	is	amazing
B-LOC	I-LOC	I-LOC	O	O

VALID! B-LOC starts, I-LOC continues

Why it works:

- **Transition matrix** blocks invalid starts
- T(O, I-LOC) = -∞ forces B-LOC first
- Considers **entire sequence jointly**
- Viterbi finds best VALID sequence

↓ ↔ ↓ ↔ ↓ ↔ ↓ ↔ ↓

(Joint prediction with transition constraints)

💡 **Key Insight:** BiLSTM learns features → CRF ensures valid sequences. Improvement: **+2-3% F1 score** on NER benchmarks!

BiLSTM-CRF: Loss Function and Training

Negative Log-Likelihood Loss

Maximize probability of correct sequence:

$$L = -\log P(y|x)$$

$$L = -s(x, y) + \log Z(x)$$

s(x,y): Score of gold sequence (easy to compute)

Z(x): Forward algorithm (expensive but tractable)

Loss Interpretation

Push up score of correct sequence s(x,y)

Push down scores of all other sequences (via Z(x))

Learnable Parameters

BiLSTM weights (emission scores)

Transition matrix T (k x k where k = num tags)

```
# PyTorch BiLSTM-CRF Training
from torchcrf import CRF

model = BiLSTM_CRF(vocab_size, tag_size, ...)
optimizer = Adam(model.parameters(), lr=1e-3)

for epoch in range(epochs):
    for batch in train_loader:
        optimizer.zero_grad()
        # Get BiLSTM emissions
        emissions = model.bilstm(batch.text)
        # CRF computes NLL loss
        loss = -model.crf(emissions, batch.tags, mask)
        loss.backward()
        clip_grad_norm_(model.parameters(), 5.0)
        optimizer.step()
```

```
# Inference with Viterbi
model.eval()
with torch.no_grad():
    emissions = model.bilstm(text)
    best_tags = model.crf.decode(emissions, mask)
```

Library: pytorch-crf

pip install pytorch-crf | Handles forward, viterbi, loss

NER Evaluation Metrics

Entity-level Metrics

Precision = TP / (TP + FP)

How many predicted entities are correct?

Recall = TP / (TP + FN)

How many gold entities are found?

F1 = 2 * P * R / (P + R)

Harmonic mean (main metric)

Strict vs Relaxed

Strict: Exact boundary + type match

Relaxed: Partial overlap OK

CoNLL uses strict evaluation

Example Evaluation

Gold: [John Smith]_PER lives in [New York]_LOC

Pred: [John]_PER lives in [New York]_LOC

"John Smith" != "John" --> FN + FP

"New York" exact match --> TP

```
# seqeval library (recommended)

from seqeval.metrics import (
precision_score, recall_score,
f1_score, classification_report)

y_true = [['O','B-PER','I-PER','O']]
y_pred = [['O','B-PER','O','O']]

f1 = f1_score(y_true, y_pred)
print(classification_report(y_true, y_pred))
```

CoNLL-2003 SOTA

BiLSTM-CRF: ~91% F1

BERT-base: ~92% F1

BERT-large: ~93% F1

Current SOTA: ~94% F1

Per-Entity F1

Report F1 for each type:

PER: 95% | LOC: 92%

ORG: 88% | MISC: 78%

Key Takeaway

Always use seqeval for entity-level F1. Token-level accuracy is misleading!

Modern NER: Transformers and Vietnamese

Transformer-based NER

BERT replaces BiLSTM as feature extractor:

Input --> BERT --> Linear --> [Softmax or CRF] --> Tags

+2-3% F1 over BiLSTM-CRF on CoNLL-2003

```
# HuggingFace BERT for NER
from transformers import (
BertForTokenClassification, Trainer)

model = BertForTokenClassification.from_pretrained(
'bert-base-cased', num_labels=num_tags)

trainer = Trainer(model=model, ...)
trainer.train()
```

BiLSTM-CRF vs BERT

BiLSTM-CRF: Fast, less data needed, ~91% F1

BERT: Better performance, slower, ~93% F1

Choose based on resources and requirements

Vietnamese NER Challenges

- **Word Segmentation:** "Hà Nội" vs "HàNội"
- **No Capitalization:** Can't use as feature
- **Compound Names:** "Nguyễn Văn A"
- **Foreign Names:** "Donald Trump"
- **Abbreviations:** "TP.HCM", "UBND"

Vietnamese NER Models

PhoBERT: Vietnamese BERT pretrained

VnCoreNLP: Word segmentation + NER

underthesea: Python Vietnamese NLP

```
from underthesea import ner
ner("Chủ tịch Nguyễn Văn A")
```

Recommendation

English: BERT + fine-tune | Vietnamese: PhoBERT or underthesea

Always preprocess with proper word segmentation!

PART 6

Lab and Assignments

Hands-on Practice

RNN/LSTM Implementation

Lab 07: Sentiment Analysis with RNN/LSTM

🎯 Lab Objectives

- Build RNN for sequence classification
- Implement LSTM with PyTorch
- Compare RNN vs LSTM performance
- Visualize training and evaluation

Dataset: IMDB Reviews

25K Train	**25K** Test	**2** Classes

Tools & Libraries

Lab Structure

1. **Data Preprocessing** — Tokenization, vocabulary, padding
2. **Build RNN Model** — Embedding + RNN + Linear layers
3. **Build LSTM Model** — Replace RNN with LSTM cell
4. **Training & Evaluation** — Train loop, accuracy, loss curves
5. **Compare & Analyze** — RNN vs LSTM performance

Expected Results: LSTM should achieve ~85-88% accuracy vs RNN's ~80-82% on longer reviews.