Neural Networks & Sequence Models
Multimodal Computational Methods in Political Science

Tamara Grechanaya [1]

[1]LMU Munich — Computational Social Science MA Program

May, 2026

Course Roadmap

1. Text as Data: Foundations & Preprocessing
2. Classical Text Classification
3. Word Embeddings & Vector Spaces
4. Document Representations & Topic Models
5. **Neural Networks & Sequence Models** ← *Today*
6. Attention & the Transformer Architecture
7. Transfer Learning & Fine-Tuning BERT

Today we cross the bridge from counting words to **learning representations**. We'll cover neural networks from scratch, build a simple neural text classifier, and meet the sequence models (RNNs, LSTMs) that finally take word order seriously. This is also where the course switches from R to Python.

Today's Outline

Part I: Why Neural Networks?

- Recap of what we've built so far
- Limits of bag-of-words + shallow models
- What neural networks add

Part II: Neural Networks from Scratch

- The neuron / dense layer
- Activation functions (ReLU, softmax)
- Forward pass
- Loss, backpropagation, optimizers

Part III: Neural Nets for Text

- Embedding layer
- Average pooling architecture
- Comparison with logistic regression

Part IV: Sequence Models

- Why word order matters
- RNNs and the vanishing gradient
- LSTMs (the gate mechanism)
- Named Entity Recognition

Part V: Practical (Python)

Table of Contents

Recap: What We've Built

In Lectures 1–4, we built a complete classical text analysis toolkit:

1. **Preprocessing**: tokenization, normalization, stop words, stemming
2. **Representation**: Document-Term Matrix, TF-IDF
3. **Classification**: Logistic Regression, Naive Bayes
4. **Word embeddings**: Word2Vec, GloVe, fastText (dense word vectors)
5. **Topic models**: LDA, STM (unsupervised discovery)

This is a remarkably powerful toolkit. For many political science research questions, classical methods are sufficient. Always try them first. But two limitations remain. Today we tackle the first; Lectures 6–7 tackle the second.

Key Concept

Limitation 1: Bag-of-words ignores word order. "The dog bit the man" and "The man bit the dog" are identical to the model.

Limitation 2: Word embeddings are static. "Bank" has one vector regardless of context.

Why Word Order Matters in Political Text

Pairs that bag-of-words cannot distinguish

- *"The opposition supported the bill"* vs. *"The bill supported the opposition"*
- *"not acceptable to the public"* vs. *"acceptable to the public, not. . . "*
- *"A vote against the climate motion"* vs. *"A motion against the climate vote"*
- *"Cuts to the NHS budget"* vs. *"Budget cuts to the NHS"* (similar but framing differs)

For tasks where word order matters — stance detection, fine-grained sentiment, named entity extraction, summarization — bag-of-words has a hard ceiling.

To break this ceiling, we need models that process sequences. Neural networks — and especially the sequence models we'll meet today — can do this.

Note: neural networks are not magic. For *many* classification tasks, a well-tuned logistic regression with TF-IDF features outperforms a hastily-built neural network. Use neural methods when you have a clear reason (sequence matters, data is abundant, classical baselines plateau).

What Neural Networks Bring

1. Learned representations
Classical methods use hand-engineered features (TF-IDF). Neural networks *learn* the right features for the task — often features no human would have designed.

2. Composition
Multiple layers stack non-linear transformations. Each layer builds on the previous one, allowing the model to capture complex patterns (interactions, context-dependence).

3. Sequence handling
RNNs, LSTMs, and Transformers process text as ordered sequences — finally moving beyond bag-of-words.

Cost: complexity

- Many more parameters (millions) than logistic regression
- Need more training data
- Need GPUs for non-trivial training
- Less interpretable: cannot read weights directly
- More hyperparameters to tune

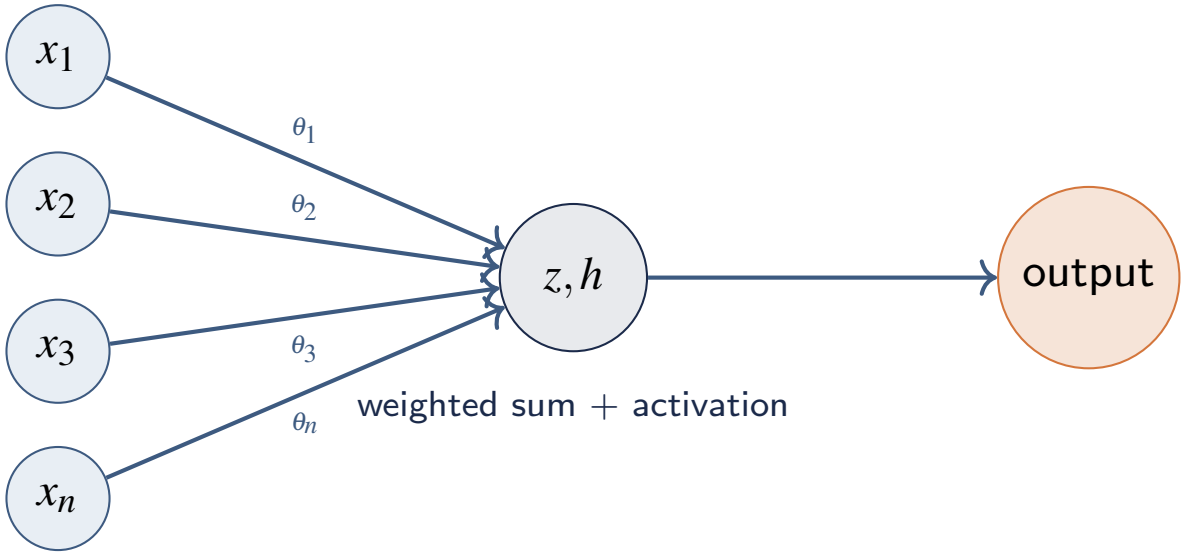
Cost: discipline

Neural methods are easier to misuse: it's easy to get a model that "works" (low training loss) but doesn't generalize. Proper train/val/test splits and regularization matter even more than in Lecture 2.

Table of Contents

The Building Block: A Single Neuron

A neuron is just a logistic regression cell. We already know this!



The two-step computation:

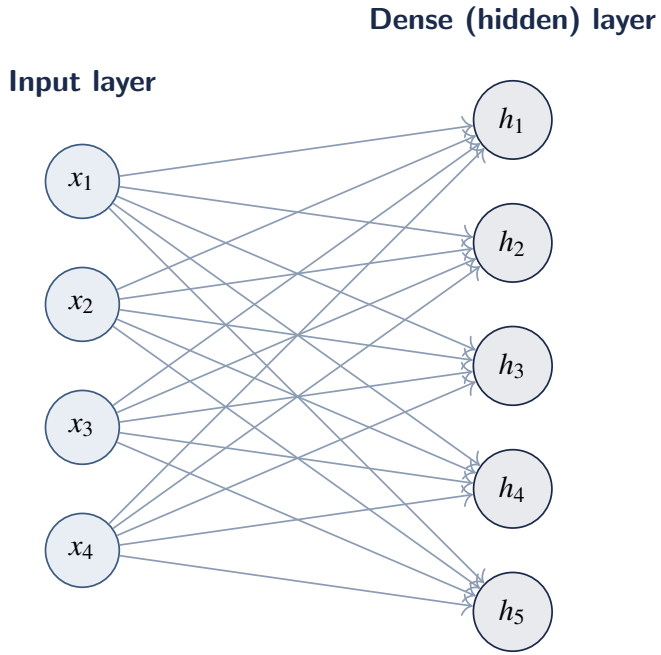
$$z = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n + b \qquad \text{(weighted sum)}$$

$$h = f(z) \qquad \text{(activation function, e.g., sigmoid or ReLU)}$$

One neuron with sigmoid activation = logistic regression. You already trained one in Lecture 2. The power of neural networks comes from **stacking many neurons in layers**.

From One Neuron to a Dense Layer

A **dense layer** (or fully connected layer) is a row of neurons that all receive the same inputs.



Mathematically: each neuron h_j has its own weights θ_j and bias b_j:

$$h_j = f\left(\sum_i \theta_{ji} x_i + b_j\right)$$

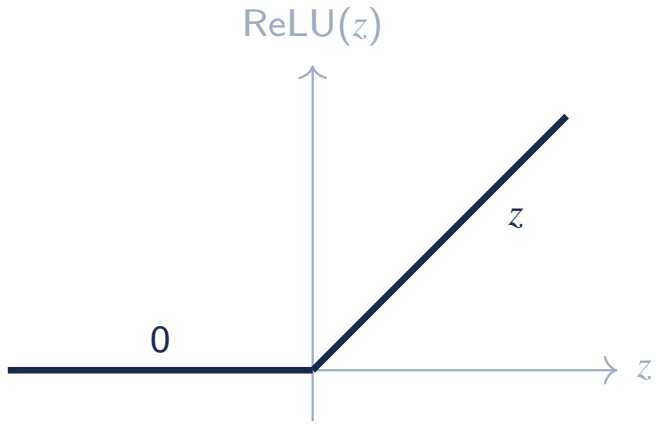
If the input has n values and the hidden layer has m neurons, the layer has $n \times m + m$ parameters. In a real NLP model, n might be 300 (embedding dim) and m might be 128. That's $\approx 38{,}000$ parameters in just one layer.

Activation Functions: ReLU

The activation function $f(\cdot)$ introduces **non-linearity** into the model.
Why we need non-linearity: without it, stacking many layers is equivalent to one layer. $\mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x}$ — still linear. We need a non-linear step in between to make the deep architecture meaningful.

ReLU (Rectified Linear Unit):

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



- Negative input $\rightarrow$ output is 0
- Positive input $\rightarrow$ output is the input itself
- Extremely simple to compute
- De facto standard for hidden layers since ≈ 2012

Other activations: sigmoid (still used in output layers for binary classification), tanh (used in LSTMs), GELU (used in BERT). For most hidden layers in modern networks: use ReLU as default.

Activation Functions: Softmax (Output Layer)

For **multi-class classification**, the output layer needs to produce a probability distribution over K classes.

Softmax converts K arbitrary real numbers into K probabilities that sum to 1:

$$\text{softmax}(z_k) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Worked example

Raw output scores from final layer: $\mathbf{z} = [2.0,\ 1.0,\ 0.1]$

Numerators: $e^{2.0} \approx 7.39$, $e^{1.0} \approx 2.72$, $e^{0.1} \approx 1.11$

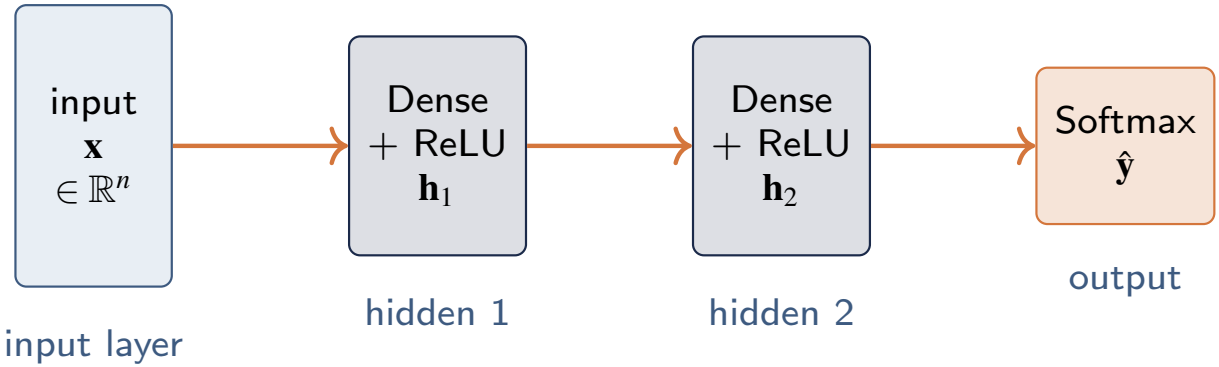
Sum: $7.39 + 2.72 + 1.11 = 11.22$

Probabilities: $\left[\frac{7.39}{11.22}, \frac{2.72}{11.22}, \frac{1.11}{11.22}\right] \approx [0.659,\ 0.242,\ 0.099]$

The class with the highest probability is the predicted class. Softmax is the multi-class generalization of sigmoid.

Forward Pass: Putting Layers Together

A neural network = a stack of dense layers with activations.



Forward pass = computing the output given the input, layer by layer:

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{h}_2 = \text{ReLU}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$

Each layer's output becomes the next layer's input. The final layer produces predictions.

Training: Loss Function

How does the network learn? Same logic as logistic regression in Lecture 2: define a **loss function** that measures how wrong the predictions are, then adjust the weights to reduce the loss.
For multi-class classification, we use **categorical cross-entropy**:

$$\mathscr{L} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}$$

For each training document i and each class k:

- $y_k^{(i)}$ is 1 if the true class is k, otherwise 0 (one-hot encoding)
- $\hat{y}_k^{(i)}$ is the predicted probability for class k
- Only the true-class term contributes: we penalize the model for assigning low probability to the correct class

Goal: find weights $\mathbf{W}_1, \mathbf{W}_2, \ldots$ that minimize $\mathscr{L}$.

This is the same idea as Lecture 2's cost function, generalized to multi-class. For binary classification it reduces to binary cross-entropy.

Training: Backpropagation (Intuition)

Backpropagation is the algorithm for computing how each weight in the network should change to reduce the loss.

The intuition:

1. **Forward pass**: compute the prediction and the loss.
2. **Backward pass**: working backward from the loss through each layer, compute how much each weight contributed to the error (this is the *gradient*).
3. **Update**: nudge each weight slightly in the direction that reduces the loss.

This is the same gradient descent idea from Lecture 2, but now applied to millions of weights organized in multiple layers. Backpropagation is just the chain rule of calculus, applied systematically layer by layer.

Key Concept

You will never implement backpropagation by hand. Modern frameworks (PyTorch, TensorFlow/Keras) compute gradients automatically via "autograd." You write the forward pass, and the framework computes the backward pass for you.

Optimizers and Learning Rate

Once we have the gradient for each weight, the simplest update rule is:

$$\theta \leftarrow \theta - \alpha \cdot \nabla \mathscr{L}$$

where α is the **learning rate**.

Choosing the learning rate:

- Too large $\rightarrow$ overshoots, training diverges
- Too small $\rightarrow$ converges very slowly
- Typical values: 10^{-3} to 10^{-5}

Modern optimizers (Adam, AdamW) adapt the learning rate per-weight based on the history of gradients. They are much more robust than plain gradient descent.

Mini-batch training:

We rarely compute the gradient on the full dataset — too slow. Instead:

1. Shuffle training data
2. Take a small **batch** (e.g., 32 documents)
3. Compute loss + gradient on the batch
4. Update weights
5. Repeat

One pass through all batches = one **epoch**. Training takes many epochs.

Default starting point: Adam optimizer, learning rate 10^{-3}, batch size 32, 10–50 epochs.

Regularization in Neural Networks

Neural networks have many more parameters than classical models — and therefore much higher risk of **overfitting**.

Three standard regularization techniques:

- **Weight decay (L2)**: same idea as Ridge from Lecture 2 — penalize large weights. Built into modern optimizers (AdamW = Adam + weight decay).
- **Dropout**: during training, randomly "turn off" a fraction (e.g., 20–50%) of neurons in each layer. The network cannot rely on any single neuron, so it learns more robust features. At inference time, all neurons are used. Dropout is the single most important neural-network regularization technique.
- **Early stopping**: train as long as validation loss decreases; stop when it starts increasing. Prevents the model from continuing to fit noise in the training data.

Caution

With small datasets (a few thousand documents), even a simple neural network can dramatically overfit. **Always use a validation set** to monitor for overfitting — don't trust training accuracy alone.

Table of Contents

The Embedding Layer

Problem: a neural network expects numerical inputs. A document is a sequence of word IDs (integers). How do we turn them into numbers the network can use?

Solution: an embedding layer. The first layer of every neural text model.

- Maintains a lookup table: one d-dimensional vector per word in vocabulary
- Input: a word ID (e.g., word #1247)
- Output: that word's d-dimensional vector (e.g., 300 numbers)

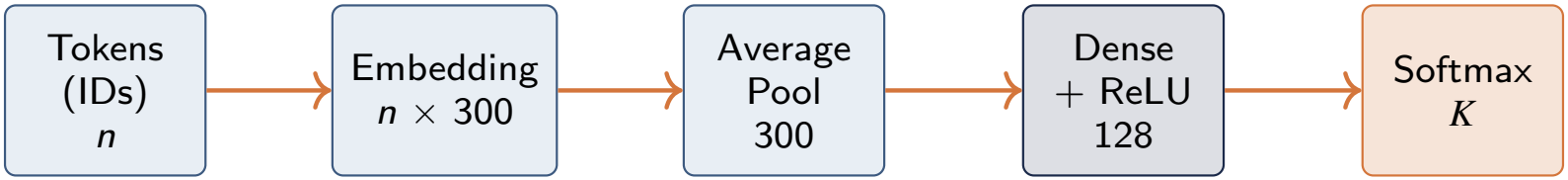
Two ways to initialize the embedding layer

- **Random initialization**: vectors start random, are learned from scratch during training. Needs lots of data.
- **Pre-trained**: initialize with GloVe / fastText vectors (Lecture 3). The network can then *fine-tune* them or keep them fixed. Works better with limited data.

In Keras: `Embedding(vocab_size, embedding_dim)`. In PyTorch: `nn.Embedding(vocab_size, embedding_dim)`.

The Simplest Neural Text Classifier

Architecture: Embedding → Average Pool → Dense → Softmax



The pipeline:

1. Each word ID is replaced by its 300-d embedding → document becomes a matrix of size $n \times 300$
2. Average pooling: collapse the matrix to a single 300-d vector by averaging across words
3. Dense layer with ReLU: transform into a 128-d representation
4. Softmax output: probabilities over K classes

This is the **minimum viable neural text classifier**. It's essentially a logistic regression on averaged embeddings, with one extra hidden layer.

Neural Classifier vs. Logistic Regression

	Logistic Regression (Lec. 2)	**Neural Classifier**
Features	TF-IDF on bag of words	Pre-trained word embeddings
Word similarity	None (each word independent)	Captured via embedding space
Training time	Seconds	Minutes–hours
Hyperparameters	1–2 (regularization λ)	10+ (layers, dim, dropout, LR, epochs...)
Interpretability	Weights = word importance	Hard to interpret directly
Data needs	Works with hundreds of docs	Needs thousands+
Best for	Strong baseline; interpretable	Complex patterns, large data

The honest truth

For **small-to-medium political text datasets** (under ~10,000 labeled documents), the simple neural network we just built often barely beats a well-tuned logistic regression with TF-IDF. The big neural-network wins come from sequence models and pre-trained transformers (BERT) — which we'll get to.

Table of Contents

The Word Order Problem (Again)

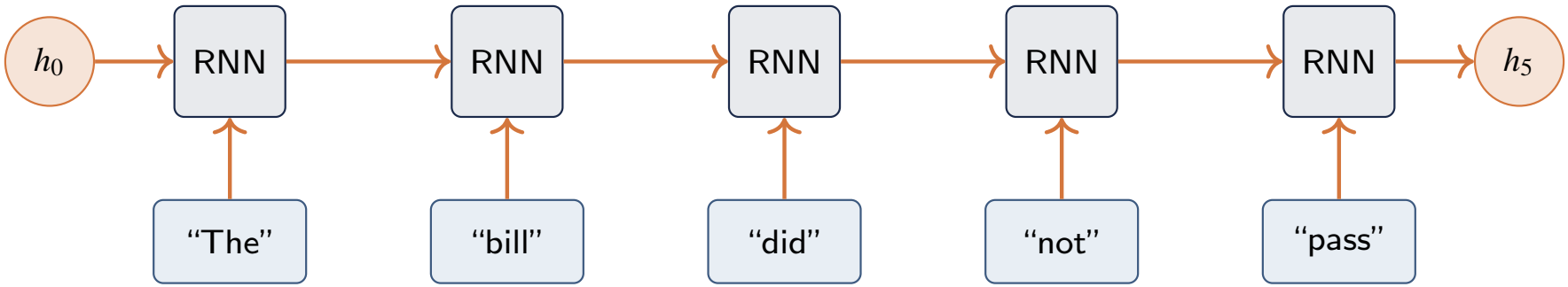
The neural classifier we just built **still ignores word order**. Average pooling collapses the sequence of word vectors into a single average — order is lost. **For many political text tasks, this is enough.** Classifying speeches by party, detecting topics, scaling by ideology — bag of words / averaged embeddings handle these well. **But some tasks genuinely require sequence:**

- **Named Entity Recognition**: "*Sunak met Macron in Paris*" — need to identify "Sunak" as PERSON, "Macron" as PERSON, "Paris" as LOCATION. The same word might be PERSON or LOCATION depending on context.
- **Sentiment with negation**: "not bad" vs. "bad" vs. "not very good"
- **Stance detection**: complex multi-clause sentences where the position depends on syntactic structure
- **Sequence labeling in general**: part-of-speech tagging, argument extraction

We need a model that reads the document word by word, keeping a memory of what it has seen.

Recurrent Neural Networks (RNNs)

An RNN processes a sequence one element at a time, maintaining a hidden state that summarizes what it has seen so far.



The recurrence: at each step t:

$$h_t = \tanh(\mathbf{W}_h h_{t-1} + \mathbf{W}_x x_t + \mathbf{b})$$

The new hidden state h_t depends on the previous hidden state h_{t-1} and the current word x_t. The **same weights** ($\mathbf{W}_h$, $\mathbf{W}_x$) are applied at every step. The final hidden state h_5 summarizes the entire sequence — and **depends on word order**.

The Vanishing Gradient Problem

In theory, RNNs can capture long-range dependencies. The hidden state carries information from the start of the sequence to the end.
In practice, vanilla RNNs struggle with sequences longer than ≈ 10–20 words.
Why? During backpropagation, gradients flow backward through every time step. The gradient at the start of the sequence is the product of many small derivatives. After 20 multiplications of numbers like 0.5:

$$0.5^{20} \approx 10^{-6}$$

The gradient **vanishes**. The network cannot effectively update weights based on early-sequence words.

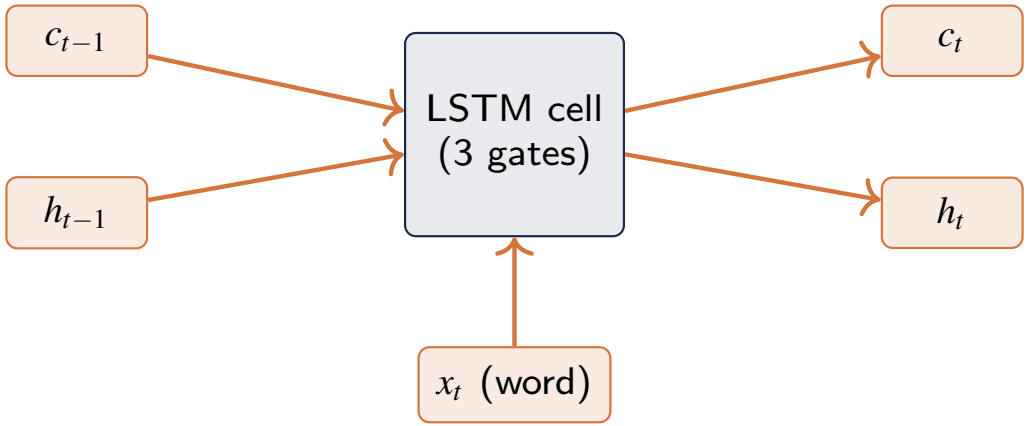
Caution

This is the same numerical underflow problem we saw with Naive Bayes log-likelihoods, but now embedded in the training dynamics. Long-range information is lost. In a political speech of 500 words, an RNN cannot easily connect the first sentence to the last.

The fix: a gated architecture that lets information flow without being multiplied at every step. Enter LSTMs.

LSTMs: The Gate Mechanism

LSTM (Long Short-Term Memory, Hochreiter & Schmidhuber, 1997) introduces a separate **cell state** c_t that flows through the sequence with minimal modification.



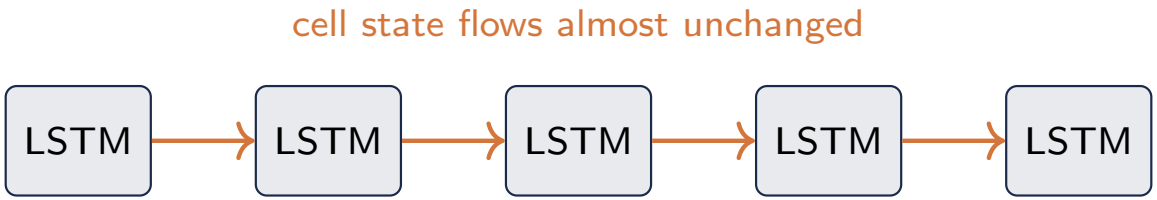
Three gates (each gate is a small neural network with sigmoid output):

- **Forget gate**: "which parts of the cell state should I forget?" — outputs values 0–1 for each cell-state dimension
- **Input gate**: "which new information should I write to the cell state?"
- **Output gate**: "which parts of the cell state should I emit as the hidden state?"

The gates are **learned** from data. The network learns when to remember, when to forget, and when to emit information — depending on the task.

Why LSTMs Solve the Vanishing Gradient Problem

The key insight: the cell state c_t flows through the sequence **additively**, not multiplicatively.
The forget gate produces values close to 1 when information should be preserved.
The cell state is updated by adding new information (input gate) without being repeatedly multiplied by small numbers.



In practice, LSTMs handle sequences of 100–500 words well. Long enough for tweets, sentences, and short paragraphs — but still struggle with full documents.

For full documents, even LSTMs fall short. That motivates the Transformer (Lecture 6) and pre-training (Lecture 7).

GRU (Gated Recurrent Unit) is a simpler variant of LSTM with 2 gates instead of 3. Comparable performance, faster training. Both are widely used.

Bidirectional LSTMs

Standard LSTMs read left-to-right. But for many tasks, the meaning of a word depends on what comes *after* it as well as before.

Why bidirectional matters

"Sunak met the ? in Paris."
Looking only at what comes before, the blank could be many things. Looking at what comes after ("in Paris"), we can guess it's likely a person/leader. Reading the sentence in both directions gives a richer representation.

Bidirectional LSTM (BiLSTM):

- Run one LSTM left-to-right
- Run another LSTM right-to-left
- Concatenate the two hidden states at each position

The result: for each word, you get a representation informed by both its left context and its right context.

BiLSTMs were the workhorse of NLP from ≈2015 to 2018. They were the state-of-the-art for tasks like NER, POS tagging, and reading comprehension before the rise of Transformers.

Named Entity Recognition with BiLSTMs

NER: identify and classify named entities in text. Crucial for extracting political actors, organizations, and locations from news, speeches, and social media.

NER output

"Sunak met Macron in Paris yesterday to discuss NATO."

Sunak	met	Macron	in	Paris	to	NATO
PERSON	O	PERSON	O	LOC	O	ORG

NER as sequence labeling:

- Input: a sequence of words
- Output: a label for each word (PERSON, LOCATION, ORGANIZATION, MISC, or O for "other")
- Architecture: word embeddings $\rightarrow$ BiLSTM $\rightarrow$ Dense $\rightarrow$ Softmax per word

Political science use cases:

- Extracting politicians, parties, institutions from news articles
- Building social-network data from co-mentions
- Geographic analysis of political coverage

In practice today: you'd use a pre-trained BERT-based NER model (spaCy, Hugging Face). But understanding the BiLSTM approach is foundational.

Limits of RNNs and LSTMs

LSTMs were transformative — but they have fundamental limits that motivate the next step.

1. No parallelism.

RNNs process the sequence one step at a time. Step t depends on step $t-1$. You cannot parallelize across time. On modern GPUs, this is extremely inefficient.

2. Long-range dependencies remain hard.

LSTMs improve over vanilla RNNs, but they still struggle with very long sequences. Information from word #1 must pass through 499 cell-state updates to reach word #500.

3. Information bottleneck.

The cell state is a fixed-size vector. The entire history of the sequence must be compressed into this single vector. Long documents lose information.

Key Concept

These problems motivate the **attention mechanism** and the **Transformer** (Lecture 6). The key idea: instead of processing the sequence one step at a time, let every position directly attend to every other position. Parallelizable and no information bottleneck.

Table of Contents

Practical Session: Switching to Python

Building a Neural Text Classifier with PyTorch

Why Python now?

- Deep learning libraries (PyTorch, TensorFlow) are Python-first
- Pre-trained models (Lecture 7) are distributed primarily for Python
- The Hugging Face ecosystem is Python-native

R alternatives exist (`keras` and `torch` packages) but you'll find more tutorials, models, and community help in Python. The transition is worth it.

What we'll build:

1. Reuse the UK House of Commons corpus from Lectures 1–2
2. Build a neural classifier (Embedding $\rightarrow$ Pooling $\rightarrow$ Dense)
3. Train it and compare to the logistic regression baseline
4. Then upgrade to an LSTM and see if it helps

Setup: Python 3.10+, PyTorch, Hugging Face `datasets`. Free GPU available via Google Colab.

Practical: Discussion Questions

1. How does the simple neural classifier's accuracy compare to the logistic regression from Lecture 2? Is the gap large enough to justify the added complexity?
2. Does the BiLSTM beat the simple averaging model? By how much? On which kinds of speeches does it help most?
3. Try initializing the embedding layer with pre-trained GloVe vectors (from Lecture 3). Does it help?
4. Try freezing the embedding layer (set `requires_grad=False`). Does the model train faster? Does accuracy change?
5. Experiment with hyperparameters: hidden dimension, dropout rate, learning rate, number of epochs. What's the cost of careless choices?

The honest takeaway: for many political text tasks at moderate scale, neural networks offer modest improvements over classical baselines. The dramatic gains come in Lecture 7 with pre-trained transformers.

Table of Contents

Key Takeaways

1. **A neural network is a stack of dense layers with non-linear activations.** Each neuron is conceptually a logistic regression; the power comes from depth and composition.
2. **Embedding layers** turn word IDs into dense vectors. Average pooling + dense layers gives a simple baseline neural text classifier.
3. **Training is gradient descent on a loss function.** Backpropagation computes gradients; optimizers (Adam) apply them. Regularization (dropout, weight decay, early stopping) prevents overfitting.
4. **RNNs** process sequences word by word, maintaining a hidden state. But they suffer from vanishing gradients on long sequences.
5. **LSTMs** use gates and a separate cell state to preserve long-range information. The workhorse of NLP from 2015–2018.
6. **Bidirectional LSTMs** read sequences in both directions for richer per-word representations — great for sequence labeling tasks like NER.
7. **Three limits** motivate the next step: no parallelism, lingering long-range issues, fixed-size information bottleneck. The Transformer fixes all three.

Home Assignment (Optional)

Task: Build a neural text classifier and compare it to your Lecture 2 baseline.
Instructions:

1. Use the same corpus and classification task as your Lecture 2 assignment (or pick a new one).
2. Implement two neural models in PyTorch:
   - Simple model: Embedding + Average Pool + Dense
   - Sequence model: Embedding + (Bi)LSTM + Dense
3. Train both, monitor validation loss, use early stopping.
4. Compare to your Lecture 2 logistic regression on accuracy, precision, recall, F1.
5. Try one variation: pre-trained embeddings, dropout rate, hidden dim, or epochs.
6. Write a **2-page report**: what worked, what didn't, what surprised you, when (if ever) neural models beat the LR baseline.

Submit: Python script (or Colab notebook) + report. Due before Lecture 6.

Readings

Required:

- Jurafsky, D. & Martin, J.H. (2025). *Speech and Language Processing*, Chapter 6: Neural Networks. https://web.stanford.edu/~jurafsky/slp3/6.pdf
- Jurafsky, D. & Martin, J.H. (2025). *Speech and Language Processing*, Chapter 13: RNNs and LSTMs. https://web.stanford.edu/~jurafsky/slp3/13.pdf

Recommended:

- Olah, C. (2015). "Understanding LSTM Networks." https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- PyTorch tutorials: https://pytorch.org/tutorials/beginner/basics/intro.html

Next week: Attention & the Transformer

The architecture that replaced RNNs and powers every modern language model