Lecture 5 — Neural Networks & Sequence Models

From Counting Words to Learning Representations

Tamara Grechanaya
Multimodal Computational Methods for Political Science

Summer Semester 2026

Contents

Learning objectives

By the end of this lecture, you should be able to:

1. Explain why bag-of-words representations and shallow classifiers have a fundamental ceiling, and what neural networks can offer beyond them.
2. Describe a neural network as a stack of dense layers with non-linear activations, and connect a single neuron back to the logistic regression from Lecture 2.
3. Explain the role of activation functions (ReLU, softmax) and why non-linearity is essential for stacking layers.
4. Walk through the forward pass and describe (intuitively) how backpropagation and gradient descent train the network.
5. Build the simplest neural text classifier: embedding → average pooling → dense → softmax.
6. Explain why bag-of-words ignores word order and what kinds of political text tasks require sequence-aware models.
7. Describe how RNNs process sequences, why they suffer from vanishing gradients, and how LSTMs solve the problem with gates.
8. Recognise when neural models are likely to beat classical baselines — and when they are not.

1 Part I — Why Neural Networks?

1.1 What we have built so far

In Lectures 1 through 4, we built a complete classical text analysis toolkit. We preprocessed raw text into tokens. We represented documents as sparse count vectors with TF-IDF weighting. We trained logistic regression and Naive Bayes classifiers, and we learned to evaluate them properly with precision, recall, F1, and confusion matrices. We discovered the world of dense word representations through Word2Vec, GloVe, and fastText. And we saw how topic models can surface latent themes in a corpus without any labels.

This toolkit is genuinely powerful. For many political science research questions — classifying speeches by party, scaling manifestos on ideological dimensions, measuring topic attention over time — classical methods are sufficient, often optimal, and almost always faster to implement than anything more complex. The first lesson of this lecture, before we start building neural networks, is: **always try the classical methods first**. If logistic regression with TF-IDF gives you 85% accuracy on your task, you may not need anything fancier.

But two limitations remain, and they motivate the rest of this course.

Limitation 1: bag-of-words ignores word order. "The bill passed" and "The bill failed" produce identical vectors. "Not acceptable" and "acceptable" look the same after stopword removal. For tasks where order matters, classical methods have a hard ceiling.

Limitation 2: word embeddings are static. The word "bank" has one vector regardless of whether it refers to a financial institution or a riverbank. Polysemy is lost.

Lecture 5 tackles the first limitation: we will build neural networks that can process sequences and finally take word order seriously. Lectures 6 and 7 tackle the second limitation, building

toward BERT-style contextual embeddings where every word gets a different vector depending on its context.

1.2 Why word order matters in political text

It is worth dwelling briefly on the word-order problem with concrete political examples, because the cost of ignoring sequence is not always obvious. Consider 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"
- "Government opposition" (opposition to the government) vs. "opposition government"

For broad-stroke classification — party prediction, topic detection, ideological scaling — bag-of-words captures enough signal that order doesn't matter much. The vocabulary of climate policy is distinctive enough that just *seeing* the climate words is informative; the order adds little. But for tasks like stance detection, fine-grained sentiment, named entity recognition, and argument structure, word order is essential. These are exactly the tasks where neural sequence models begin to pay off.

1.3 What neural networks add (and what they cost)

Neural networks bring three things to text analysis beyond what classical methods offer.

First, they **learn representations**. Classical methods use hand-engineered features — you decide that words should be counted with TF-IDF weights, that bigrams should be included, that rare words should be dropped. Neural networks learn the right features for the task automatically, often discovering patterns no human would have hand-designed.

Second, they enable **composition**. By stacking multiple layers of non-linear transformations, a neural network can capture interactions and context-dependent meanings. The first layer might learn that certain word combinations matter; the second layer might learn that those combinations are reliable indicators of stance only in certain contexts; the third layer might learn yet higher-level abstractions.

Third, they can **handle sequences**. Recurrent neural networks, LSTMs, and Transformers process text as ordered sequences — finally going beyond bag-of-words.

The costs are real. Neural networks have millions of parameters, so they need much more training data — typically tens of thousands of labeled examples. Training takes minutes or hours and is often infeasible without a GPU. Hyperparameter tuning is more involved: instead of one regularization parameter, you now have many choices (number of layers, layer sizes, dropout rate, learning rate, batch size, number of epochs). And interpretability suffers: the weight matrices of a neural network do not have any direct interpretation in the way a logistic regression's coefficients do.

One further risk: neural methods are **easier to misuse** than classical methods. It is straightforward to get a model that "works" on your training data but generalises poorly to new examples. The discipline of train/validation/test splits, careful regularisation, and honest evaluation matters even more here than in Lecture 2.

2 Part II — Neural Networks from Scratch

2.1 The building block: a single neuron

The most important conceptual move of this lecture is also the simplest: a single neuron in a neural network is essentially the same thing as a logistic regression. You already understand it.

A neuron takes a vector of inputs $x_1, x_2, \ldots, x_n$, computes a weighted sum with bias, and applies a non-linear function:

$$z = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n + b \tag{1}$$
$$h = f(z) \tag{2}$$

The first line is exactly the linear combination from Lecture 2's logistic regression. The weights θ_i play the same role as the regression coefficients; the bias b is the intercept. The second line applies an *activation function* f to squash the weighted sum into a more useful range. If f is the sigmoid function, a single neuron *is* logistic regression.

The power of neural networks does not come from any individual neuron — it comes from stacking many of them in layers and using non-linear activations between layers.

2.2 From one neuron to a dense layer

A **dense layer** (also called a fully connected layer) is a row of neurons that all receive the same inputs but each have their own weights. If the input is a vector of n numbers and the layer has m neurons, the layer produces a vector of m outputs:

$$h_j = f\left(\sum_{i=1}^{n} \theta_{ji}\, x_i + b_j\right) \quad \text{for } j = 1, 2, \ldots, m \tag{3}$$

The full set of weights forms a matrix of size $m \times n$, plus a bias vector of size m. Total parameters: $m \times n + m$. To make this concrete: if the input is a 300-dimensional word embedding ($n = 300$) and the layer has 128 neurons ($m = 128$), the layer has $128 \times 300 + 128 = 38{,}528$ parameters. That is far more than a logistic regression, and it's just one layer.

2.3 Activation functions: ReLU and the need for non-linearity

Without non-linearity, depth gives you nothing. Imagine stacking two dense layers without any activation in between:

$$\mathbf{h}_2 = \mathbf{W}_2 \mathbf{h}_1 = \mathbf{W}_2(\mathbf{W}_1 \mathbf{x}) = \mathbf{W}'\mathbf{x} \tag{4}$$

The composition $\mathbf{W}_2\mathbf{W}_1$ is just another matrix. Without non-linearity, a stack of layers collapses into a single linear transformation.

The most common activation function in modern neural networks is the **ReLU** (Rectified Linear Unit):

$$\mathrm{ReLU}(z) = \max(0, z) \tag{5}$$

If the input z is negative, the output is zero; if positive, the output equals the input. It is extremely simple to compute, works well in practice, and has been the de facto default for hidden layers since around 2012. Use it unless you have a specific reason not to.

Other activation functions exist: sigmoid is still used in output layers for binary classification; tanh is used inside LSTM cells; GELU is used in BERT. But for hidden layers, ReLU is the standard.

2.4 Activation functions: softmax for multi-class output

When the task is multi-class classification — classifying parliamentary speeches into Conservative, Labour, LibDem, SNP, or Green — the output layer must produce a probability distribution over the K classes. The **softmax** function does this:

$$\text{softmax}(z_k) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \tag{6}$$

Softmax takes K raw scores (logits) and converts them into K probabilities that sum to 1.

As a worked example, suppose the final layer produces three raw scores $\mathbf{z} = [2.0,\ 1.0,\ 0.1]$. Computing the exponentials gives $e^{2.0} \approx 7.39$, $e^{1.0} \approx 2.72$, $e^{0.1} \approx 1.11$, with sum 11.22. The probabilities are approximately $[0.659,\ 0.242,\ 0.099]$. The class with the highest probability (here, class 1 at 0.659) is the predicted class.

2.5 The forward pass

A neural network is a stack of dense layers with non-linear activations between them. Computing the prediction from an input $\mathbf{x}$ is called the **forward pass**. Each layer's output becomes the next layer's input:

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{7}$$

$$\mathbf{h}_2 = \text{ReLU}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2) \tag{8}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3) \tag{9}$$

This is a three-layer network: two hidden layers and one output layer. "Deep" neural networks simply stack more such layers.

2.6 Training: the loss function

How does the network learn the right weights? Same idea 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, the standard loss is **categorical cross-entropy**:

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)} \tag{10}$$

This is the same as Lecture 2's binary cross-entropy, generalised to K classes. For each training document i, the true label is a one-hot vector: a vector of length K that is 1 at the position of the true class and 0 elsewhere. Only the true-class term contributes to the loss, and that term

is the negative log of the predicted probability for the correct class. If the model assigns high probability to the correct class, the loss is small; if it assigns low probability, the loss is large.

2.7 Training: backpropagation (intuition only)

Backpropagation is the algorithm that computes how much each weight should change to reduce the loss. It works in three phases:

1. **Forward pass.** Compute the prediction and the loss.
2. **Backward pass.** Working backward through each layer, compute the gradient for each weight: how much it contributed to the error.
3. **Update.** Nudge each weight slightly in the direction that reduces the loss.

The mathematical machinery is the chain rule from calculus, applied systematically layer by layer. Each weight gets blame proportional to how much it contributed to the error. In practice, you will never implement backpropagation by hand. Modern deep learning frameworks (PyTorch, TensorFlow) compute gradients automatically through *automatic differentiation*.

2.8 Optimisers and learning rate

Once we have the gradients, the simplest update is gradient descent:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \cdot \nabla \mathcal{L} \tag{11}$$

where α is the **learning rate** — the step size. Choosing it matters: too large and the optimisation overshoots; too small and it converges very slowly. Typical values are between 10^{-3} and 10^{-5}.

Modern optimisers like **Adam** and **AdamW** are far more robust than plain gradient descent. They adapt the learning rate per-weight based on the history of gradients. The default starting point in practice is: AdamW optimiser, learning rate 10^{-3}, batch size 32, train for 10–50 epochs.

In practice, we don't compute gradients on the full dataset — that would be too slow. Instead, we use **mini-batch** training: take a small batch (e.g., 32 documents), compute the loss and gradients on that batch, update the weights, and repeat. One pass through all batches is one **epoch**.

2.9 Regularisation

Neural networks have many more parameters than classical models, so they overfit much more easily. Three standard regularisation techniques are essential.

Weight decay (L2) adds a penalty proportional to the squared magnitude of the weights, keeping them small. This is the same idea as Ridge regression from Lecture 2, and is built into AdamW.

Dropout randomly "turns off" a fraction of neurons in each layer during training (typically 20–50%). The network cannot rely on any specific neuron, so it learns more robust, distributed features. At inference time, all neurons are used.

Early stopping is the simplest and arguably most effective technique. Train as long as the validation loss is decreasing; stop as soon as it starts increasing. This prevents the model from fitting noise in the training data after the useful patterns have been learned.

With small datasets (a few thousand documents), even a simple neural network can overfit dramatically. Always monitor validation accuracy during training; training accuracy alone is meaningless.

3 Part III — Neural Networks for Text

3.1 The embedding layer

Neural networks expect numerical input. A document, after tokenisation, is a sequence of word IDs — integers indicating positions in the vocabulary. The **embedding layer** turns these integers into vectors. It maintains a lookup table: one d-dimensional vector per word in the vocabulary. The input is a word ID (say, word 1247), and the output is that word's d-dimensional vector.

Conceptually, the embedding layer is a $|V| \times d$ matrix, where $|V|$ is the vocabulary size and d the embedding dimension. "Looking up" an embedding is just selecting the appropriate row.

There are two initialisation strategies. **Random initialisation** starts with random vectors learned from scratch during training. This works well with large datasets but is wasteful for smaller ones. **Pre-trained initialisation** uses GloVe or fastText vectors from Lecture 3 as the starting point, either keeping them fixed (the embedding layer doesn't update during training) or *fine-tuning* them alongside the rest of the model. Pre-trained initialisation is almost always better when training data is limited.

3.2 The simplest neural text classifier

The minimum viable neural text classifier has four components:

1. **Embedding layer.** Each word ID is replaced by its d-dimensional vector. A document of n words becomes a matrix of size $n \times d$.
2. **Average pooling.** Collapse the matrix into a single d-dimensional vector by averaging across words.
3. **Dense layer with ReLU.** Transform the pooled vector into a different representation (say, 128-dimensional).
4. **Softmax output.** Produce probabilities over the K classes.

This is essentially a logistic regression on averaged word embeddings plus one extra hidden layer. It is the first thing to try when you want a neural baseline for text classification.

3.3 Neural classifier vs. logistic regression: the honest comparison

How does the simple neural classifier compare to the logistic regression from Lecture 2? On most political text classification tasks with moderate-sized datasets (a few thousand to tens of thousands of labeled documents), the answer is sobering: **the simple neural classifier often barely beats a well-tuned logistic regression with TF-IDF**.

This is not a failure of neural networks. It reflects what each method captures. Logistic regression with TF-IDF captures word-level signal very efficiently. For tasks where word-level signal is most of what matters (party classification, topic detection, sentiment on long texts), the marginal benefit of a small neural network is small. The benefits grow when you have large training sets, when word order matters, when you use sequence models rather than averaging, or when you

use pre-trained transformer models (Lecture 7). The dramatic gains in modern NLP come from that last item.

4 Part IV — Sequence Models

4.1 The word order problem (revisited)

The simple neural classifier described above still ignores word order. Average pooling collapses the sequence of word vectors into a single average; the order in which words appeared is gone. For tasks where order matters — stance detection, NER, sentiment with negation — we need a model that processes words *in sequence*, maintaining some kind of memory of what it has seen so far. This is the role of **recurrent neural networks** (RNNs) and their improved variants (LSTMs, GRUs).

4.2 Recurrent neural networks

An RNN processes a sequence one element at a time, maintaining a **hidden state** that summarises what it has seen so far. At each step t, the RNN takes the current word x_t and the previous hidden state h_{t-1}, and produces a new hidden state h_t:

$$h_t = \tanh(\mathbf{W}_h h_{t-1} + \mathbf{W}_x x_t + \mathbf{b}) \tag{12}$$

Three things to notice. First, the same weights $\mathbf{W}_h$ and $\mathbf{W}_x$ are used at every time step — the model is *recurrent*. Second, the new hidden state depends on both the current word and the previous hidden state, giving the model memory. Third, after processing the entire sequence, the final hidden state h_n summarises the whole document, and crucially it depends on the order in which words appeared.

For a classification task, feed h_n into a final dense layer with softmax. For sequence labelling tasks such as NER, produce one prediction per word using each h_t.

4.3 The vanishing gradient problem

In theory, RNNs can capture long-range dependencies. In practice, vanilla RNNs struggle with sequences longer than about 10 to 20 words.

The problem lies in training. Gradients flow backward through every time step. The gradient for a weight affecting an early word must pass through many multiplications — one per time step. If those multiplications involve numbers less than 1, the gradient shrinks exponentially. Concretely: if each step multiplies the gradient by roughly 0.5, after 20 steps the gradient is $0.5^{20} \approx 10^{-6}$. The signal has effectively **vanished**. The network cannot update its weights based on information from the start of the sequence.

This is the same numerical underflow issue we saw with Naive Bayes log-likelihoods in Lecture 2, but embedded in the training dynamics rather than in inference. The practical consequence is that vanilla RNNs cannot connect the first sentence of a 500-word speech to the last.

4.4 LSTMs: the gate mechanism

The LSTM (Long Short-Term Memory), introduced by Hochreiter and Schmidhuber in 1997, solves the vanishing gradient problem with two ideas: a separate **cell state** that flows through the sequence with minimal modification, and **gates** that control what information enters and leaves the cell state.

An LSTM cell has three gates, each implemented as a small neural network with sigmoid output (values between 0 and 1):

- **Forget gate.** Decides which parts of the cell state should be discarded. Output close to 1 retains the information; close to 0 forgets it.
- **Input gate.** Decides which new information from the current word and previous hidden state should be written to the cell state.
- **Output gate.** Decides which parts of the updated cell state should be emitted as the hidden state for this time step.

The gates are learned from data. The network learns when to remember, when to forget, and when to emit information, depending on the task. For sentiment classification, it might learn to keep "not" in the cell state for many steps so it can flip the sentiment of a later word. For NER, it might learn to hold context that helps disambiguate a name.

4.5 Why LSTMs solve the vanishing gradient problem

The key insight is that the cell state in an LSTM flows through the sequence *additively*, not multiplicatively. New information is added to the cell state (gated by the input gate), and old information is retained unchanged when the forget gate is close to 1. The gradient does not get multiplied by small numbers at every step. When the forget gate is open, the gradient flows through with minimal attenuation.

In practice, LSTMs handle sequences of 100 to 500 words well — long enough for tweets, sentences, and short paragraphs. They were the workhorse of NLP from roughly 2015 to 2018, achieving state-of-the-art results on tasks including machine translation, summarisation, NER, and reading comprehension. A simpler variant called the **GRU** (Gated Recurrent Unit) achieves comparable performance with only two gates instead of three.

4.6 Bidirectional LSTMs

Standard LSTMs read left-to-right: the hidden state at position t only knows about words 1 through t. But for many tasks, the meaning of a word depends on what comes *after* it as well as before.

Consider the sentence "Sunak met the _ in Paris." Looking only at the preceding context, the blank could be almost anything. But the following context ("in Paris") strongly suggests another person or leader. For NER specifically, the right context is just as informative as the left context.

The **bidirectional LSTM (BiLSTM)** runs two LSTMs in parallel: one left-to-right and one right-to-left. At each position, you get two hidden states (one from each direction), concatenated into a single rich representation. Position t thus has access to both the prefix and the suffix of the sequence. BiLSTMs were the dominant architecture for sequence labelling tasks from around 2015 until the rise of Transformers.

4.7 Named entity recognition with BiLSTMs

Named Entity Recognition is the task of identifying and classifying named entities (people, organisations, locations, and so on) in text. It is one of the most important sequence labelling tasks for political science applications.

As a concrete example, consider the sentence "*Sunak met Macron in Paris yesterday to discuss NATO.*" The model should assign one label per word:

Sunak	met	Macron	in	Paris	yesterday	to	discuss	NATO
PER	O	PER	O	LOC	O	O	O	ORG

The architecture for NER with BiLSTMs is: word embeddings → BiLSTM → dense layer (per word) → softmax (per word), producing a probability distribution over entity types for each token.

Political science use cases for NER include extracting politicians, parties, and institutions from news articles; building co-mention networks of political actors; analysing the geographic focus of political coverage; and tracking which actors appear together in legislative debates. Today, the practical recommendation is to use a pre-trained transformer-based NER model from libraries like spaCy or Hugging Face. Understanding the BiLSTM approach is nonetheless foundational, and many older systems still use it.

4.8 The limits of RNNs and LSTMs

LSTMs were transformative, but they have three fundamental limitations that motivate the next step in the course.

No parallelism. RNNs process the sequence one step at a time; step t depends on step $t-1$. You cannot parallelise across time. On modern GPUs, which excel at parallel computation, this is extremely inefficient.

Long-range dependencies remain hard. LSTMs are far better than vanilla RNNs, but they are not perfect. Information from word 1 must still pass through many cell-state updates to reach word 500. For very long documents, even LSTMs struggle.

Information bottleneck. The cell state is a fixed-size vector. The entire history of the sequence must be compressed into this single vector. As sequences get longer, more information must be squeezed through the bottleneck, and some is lost.

These three limitations motivate the **attention mechanism** and the **Transformer architecture** (Lecture 6). The key idea: instead of processing the sequence one step at a time, let every position in the sequence directly attend to every other position. This is fully parallelisable, has no information bottleneck, and handles arbitrary-length dependencies. It is the architecture behind every modern large language model.

5 Part V — Limitations and the practical session

5.1 When neural models beat classical baselines (and when they don't)

Before turning to the practical, it is worth making the honest comparison explicit. For many political text tasks at moderate scale, neural networks offer only modest improvements over classical methods. The rule of thumb is:

- Classical methods (logistic regression, TF-IDF) often match or beat small neural networks when the dataset is under ∼10,000 labeled documents.
- Neural sequence models (BiLSTMs) start to pull ahead on tasks where word order matters (stance, fine-grained sentiment, NER).
- Pre-trained transformer models (Lecture 7) consistently and substantially beat everything else — at the cost of much greater computational complexity.

The key discipline throughout is the same as in Lecture 2: always compare against a well-tuned classical baseline before claiming that a more complex model is better.

5.2 The practical session

This lecture's practical session marks the transition from R to Python. The deep learning ecosystem — PyTorch, TensorFlow/Keras, Hugging Face Transformers — is Python-first. R alternatives exist (`keras` and `torch`), but you will find vastly more tutorials, models, and community support in Python. We will use **PyTorch**, which has become the de facto standard for research. The complete code is provided in a separate script (`Lecture5_Practical.py`) and is designed to run on a laptop CPU.

The workflow proceeds in three phases. First, we prepare the UK House of Commons data from Lectures 1–2 for PyTorch by building a tokeniser, a vocabulary, and a custom Dataset class. Second, we build and train the simple neural classifier (embedding → average pooling → dense → softmax) and compare it against the Lecture 2 logistic regression baseline. Third, we upgrade to a BiLSTM and compare again. Throughout, we instrument validation accuracy and monitor for overfitting.

The discussion questions at the end of the practical focus on the things most worth internalising: how much the neural classifier outperforms logistic regression (and whether that gap justifies the added complexity), whether pre-trained embeddings help, how sensitive the results are to hyperparameter choices, and on which kinds of speeches the BiLSTM gains most over simple averaging.

Key takeaways

1. **Always try classical methods first.** For many political text tasks at moderate scale, logistic regression with TF-IDF is competitive with simple neural networks. Neural methods shine with large datasets, sequence-aware tasks, and pre-trained transformers.

2. **A neural network is a stack of dense layers with non-linear activations.** Each neuron is conceptually identical to logistic regression; the power comes from depth, composition, and non-linearity.

3. **The embedding layer** turns word IDs into dense vectors. Average pooling plus dense layers gives the simplest neural text classifier — a useful baseline.

4. **Training is gradient descent on a loss function.** Backpropagation computes gradients automatically; modern optimisers (Adam, AdamW) apply them. Regularisation (dropout, weight decay, early stopping) is essential.

5. **RNNs process sequences word by word**, maintaining a hidden state. Vanilla RNNs suffer from vanishing gradients and cannot capture long-range dependencies.

6. **LSTMs use gates and a separate cell state** to preserve long-range information. They were the workhorse of NLP from 2015 to 2018, and remain useful today.

7. **Bidirectional LSTMs** read sequences in both directions, producing richer per-word representations — the standard architecture for sequence labelling tasks such as NER.

8. **Three limitations of RNNs/LSTMs** motivate the Transformer: no parallelism, lingering long-range limitations, and a fixed-size information bottleneck. Lecture 6 addresses all three.

Required reading

- 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 reading

- Olah, C. (2015). "Understanding LSTM Networks." https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (The single best intuitive explanation of LSTM gates on the internet, with clear diagrams.)
- PyTorch official tutorials. https://pytorch.org/tutorials/beginner/basics/intro.html

Looking ahead

In **Lecture 6**, we move from RNNs and LSTMs to the architecture that has dominated NLP since 2017: the **Transformer**. The key innovation is the *attention mechanism*, which lets every position in a sequence directly attend to every other position — removing the information bottleneck of RNNs and enabling massive parallelism. By the end of Lecture 6, you will understand how self-attention works, why multi-head attention helps, and how the full Transformer architecture is assembled. This sets up Lecture 7, where we fine-tune BERT on a political classification task — the practical culmination of the course.