Lecture 5 Companion: Neural Networks and Neural Language Models

From Intuition to Matrix Mechanics

NLP Course Companion

Contents

1 The Neural Shift: From Counting to Understanding

In traditional Natural Language Processing (NLP), the primary method for predicting the next word was the **N-Gram Model**.

How N-Grams Work: They are purely statistical. They count how often a specific sequence of words (an N-gram) appears in the training data. If "the cat sat on" was followed by "the mat" 100 times and "the chair" 10 times, the model predicts "mat" as more likely.

1.1 The Limitations of N-Grams

While foundational, N-grams face critical limitations:

1. **Sparsity (The "Never Seen It" Problem):** If a specific sequence of words (e.g., "my alligator ate homework") never appeared in the training data, the N-gram model assigns it a probability of zero. It deems the sequence impossible, even if it is perfectly valid English.
2. **Lack of Generalization (The Synonym Problem):** N-grams treat every word as an isolated token. "Cat" and "dog" are as different to an N-gram model as "cat" and "philosophy." The model cannot generalize knowledge; it doesn't understand that contexts suitable for "cat" are likely suitable for "dog."

1.2 The Neural Solution: Embeddings

Neural Language Models (NLMs) overcome these issues by changing how words are represented. Instead of exact counting, they use **Embeddings** (distributed representations).

An embedding is a dense vector (a list of numbers) that captures the meaning of a word.

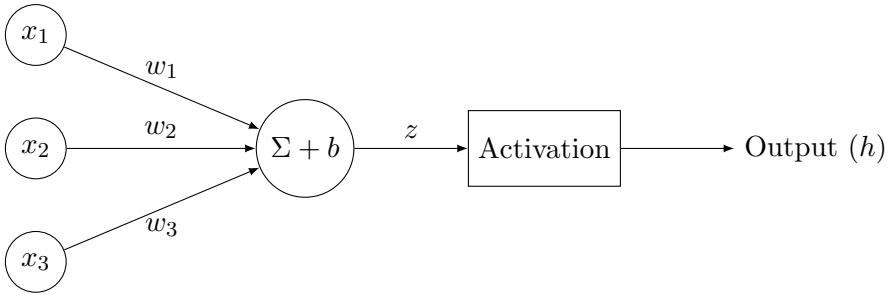
- *Cat* might be `[0.5, -0.2, 0.1, ...]`
- *Dog* might be `[0.6, -0.1, 0.1, ...]`

Intuition: The GPS of Meaning

Embeddings are like coordinates. Words with similar meanings or uses are placed close together in this vector space. Because the vectors for "cat" and "dog" are similar, the neural network learns mathematical rules that apply similarly to both, allowing it to generalize.

2 The Building Block: The Artificial Neuron

The entire network is built from artificial neurons. A neuron takes several inputs and produces a single output in two crucial steps:



2.1 Step 1: Linear Transformation (Weighing the Evidence)

Each input (x_i) is multiplied by a corresponding weight (w_i). The neuron sums these weighted inputs and adds a bias (b).

$$z = (w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n) + b$$

- **Weights (W):** Determine the importance of each input.
- **Bias (b):** The neuron's "default leaning" or baseline activation if all inputs are zero.

2.2 Step 2: Non-Linear Activation (Making the Decision)

The result z is passed through an **Activation Function** (e.g., Sigmoid, ReLU).

$$h = \text{Activation}(z)$$

Why is this crucial? If we only used linear transformations, the entire network—no matter how deep—would mathematically collapse into a single linear equation. It could only learn straight-line relationships. Real-world data, especially language, is complex and non-linear. The activation function introduces the "bends" and "curves" the network needs to model reality.

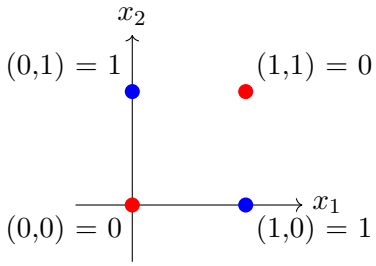
3 Why "Deep" Learning? The XOR Paradox

Why do we need "hidden layers" between the input and output? Why not just connect them directly?

The classic proof of necessity is the **XOR (Exclusive OR) Problem**.

The Task: Output 1 if inputs are different (0,1 or 1,0). Output 0 if inputs are the same (0,0 or 1,1).

The Failure of Simple Networks: A simple network without hidden layers is a linear classifier. It tries to draw a single straight line to separate the data. If you plot the XOR data, this is impossible:



You cannot draw one straight line that separates the 1s (blue) from the 0s (red). It is **linearly inseparable**.

The Deep Solution: Transforming the Space By adding a **Hidden Layer**, the network doesn't just try to draw a line. Instead, it *transforms the input space*. The hidden layer warps the data, projecting it into a new representation where the classes *become* linearly separable.

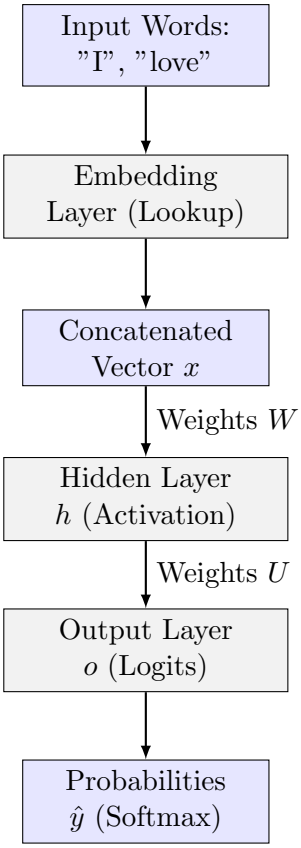
Intuition: Warping the Space

Imagine the points are on a flat piece of paper. The hidden layer is like folding or bending the paper in 3D space so that the points line up perfectly, allowing a flat plane (a line in higher dimensions) to separate them.

4 Architecture of a Neural Language Model (NLM)

A simple Feedforward NLM predicts the next word w_t based on a fixed window of context words.

Example: Context = "I love", Target = "NLP".



1. **Input/Embedding:** Context words are converted to embeddings.
2. **Concatenation:** Embeddings are combined into the input vector x.
3. **Hidden Layer:** x is processed using weights W and an activation function to produce the hidden state h.
4. **Output Layer:** h is processed using weights U to produce raw scores (logits) o for every word in the vocabulary.
5. **Softmax:** Logits are converted into a probability distribution $\hat{y}$.

5 The Mechanics of Learning: A Mathematical Walkthrough

How does the network adjust the weights (W and U) to improve? We'll use a concrete example to illustrate both the **Scalar View** (step-by-step arithmetic) and the **Matrix View** (how computers efficiently do this).

5.1 Scenario Setup

- **Task:** Predict the word after "I love". **Target:** "NLP".
- **Vocabulary (V=3):** ["I" (index 0), "love" (index 1), "NLP" (index 2)]
- **Dimensions:** Embedding size (d=2), Hidden layer size (h=2).
- **Learning Rate (η):** 0.1 (Used during updates).

Initialized Matrices (Randomly set)

Embeddings (E):

- "I" = [0.1, 0.2]
- "love" = [0.3, 0.4]

Input-Hidden Weights (W): (Shape: 2 x 4)

$$W = \begin{bmatrix} 0.5 & 0.2 & 0.1 & 0.1 \\ 0.1 & 0.3 & 0.5 & 0.2 \end{bmatrix}$$

Hidden-Output Weights (U): (Shape: 3 x 2)

$$U = \begin{bmatrix} 0.2 & 0.1 \\ 0.4 & 0.3 \\ 0.6 & 0.5 \end{bmatrix}$$

(Note: We will ignore biases for simplicity in this example.)

5.2 Part 1: The Forward Pass (Making a Prediction)

We push the input through the network to see what it predicts.

Step 1: Create Input Vector (x)

We look up the embeddings for "I" and "love" and concatenate them.

Scalar View: $x = [0.1, 0.2, 0.3, 0.4]$

Matrix View

A column vector x (Shape 4x1).

$$x = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix}$$

Step 2: Calculate Hidden Layer Pre-Activation (z)

We multiply the input vector by the weights matrix W. $z = Wx$.

Scalar View:

$$\text{Neuron 1 } (z_1) = (0.5 \cdot 0.1) + (0.2 \cdot 0.2) + (0.1 \cdot 0.3) + (0.1 \cdot 0.4) = 0.16$$
$$\text{Neuron 2 } (z_2) = (0.1 \cdot 0.1) + (0.3 \cdot 0.2) + (0.5 \cdot 0.3) + (0.2 \cdot 0.4) = 0.30$$

Matrix View

$$\begin{bmatrix} 0.5 & 0.2 & 0.1 & 0.1 \\ 0.1 & 0.3 & 0.5 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.16 \\ 0.30 \end{bmatrix}$$

Step 3: Apply Activation (h)

We apply the Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$. $h = \sigma(z)$.

Scalar View: $h_1 = \sigma(0.16) \approx 0.54$
$h_2 = \sigma(0.30) \approx 0.57$

Matrix View

$$h = \begin{bmatrix} 0.54 \\ 0.57 \end{bmatrix}$$

Step 4: Calculate Output Logits (o)

We multiply the hidden state h by the output weights matrix U. $o = Uh$.

Scalar View:

$$\text{Score for "I" } (o_1) = (0.2 \cdot 0.54) + (0.1 \cdot 0.57) = 0.165$$
$$\text{Score for "love" } (o_2) = (0.4 \cdot 0.54) + (0.3 \cdot 0.57) = 0.387$$
$$\text{Score for "NLP" } (o_3) = (0.6 \cdot 0.54) + (0.5 \cdot 0.57) = 0.609$$

Matrix View

$$\begin{bmatrix} 0.2 & 0.1 \\ 0.4 & 0.3 \\ 0.6 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.54 \\ 0.57 \end{bmatrix} = \begin{bmatrix} 0.165 \\ 0.387 \\ 0.609 \end{bmatrix}$$

Step 5: Probabilities (Softmax)

We convert the logits into probabilities by exponentiating and normalizing.

Scalar View:

1. Exponentiate: $e^{0.165} \approx 1.18$, $e^{0.387} \approx 1.47$, $e^{0.609} \approx 1.84$
2. Sum (Normalizer): $1.18 + 1.47 + 1.84 = 4.49$
3. Normalize: P("NLP") $= 1.84 \ / \ 4.49 \approx 0.41$

Matrix View

The prediction vector $\hat{y}$.

$$\hat{y} = \begin{bmatrix} 0.26 \\ 0.33 \\ 0.41 \end{bmatrix}$$

5.3 Part 2: The Backward Pass (Learning from Mistakes)

The model predicted 41% for "NLP", but the target is 100%. We need to adjust the weights using **Backpropagation** and **Gradient Descent**.

Intuition: Gradient Descent

We calculate the gradient (the slope) of the Loss (the error) with respect to each weight. We then take a small step (the learning rate η) in the opposite direction of the gradient to reduce the loss.

Step 1: Calculate the Error Signal (δ_o)

We use **Cross-Entropy Loss**. When combined with Softmax, the derivative (the error signal we need) is remarkably simple:

$$\delta_o = \text{Prediction} - \text{Target} = \hat{y} - y$$

The true target y is a "one-hot" vector where "NLP" is 1.

Scalar View: Error "NLP": $0.41 - 1 = -0.59$

Matrix View

$$\delta_o = \begin{bmatrix} 0.26 \\ 0.33 \\ 0.41 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.26 \\ 0.33 \\ -0.59 \end{bmatrix}$$

The negative error means we need to increase the score for "NLP".

Step 2: Update Output Weights (U)

We need the gradient of the Loss with respect to U (∇U).

The Golden Rule of Weight Updates (Derived from the Chain Rule)

Gradient = Error at the Destination $\times$ Input from the Source

Here, Destination = Output layer (δ_o); Source = Hidden layer (h).

Scalar View: Update the weight connecting h_1 to "NLP" ($U_{3,1}$, currently 0.6). Gradient = Error at "NLP" $\times$ Input from h_1
Gradient $= -0.59 \times 0.54 = -0.318$

Update Rule: $U_{\text{new}} = U_{\text{old}} - (\eta \times \text{Gradient})$
$U_{\text{new}} = 0.6 - (0.1 \times -0.318) = 0.6 + 0.0318 = 0.6318$

Matrix View

We use the **Outer Product**: $\nabla U = \delta_o \cdot h^T$.

$$\nabla U = \begin{bmatrix} 0.26 \\ 0.33 \\ -0.59 \end{bmatrix} \cdot \begin{bmatrix} 0.54 & 0.57 \end{bmatrix} \approx \begin{bmatrix} 0.14 & 0.15 \\ 0.18 & 0.19 \\ -0.32 & -0.34 \end{bmatrix}$$

Step 3: Backpropagate Error to the Hidden Layer (δ_h)

This is the core of backpropagation. We need to know how much the hidden layer was responsible for the final error before we can update W.

Intuition: The Blame Game

Hidden neuron h_1 connected to all three outputs. How much blame does h_1 deserve? We calculate h_1's total blame by summing the errors it contributed to (δ_o), weighted by the strength of the connections (U).

Calculating δ_h involves two parts:

Part A: The Weighted Error

We pull the error back through U. Mathematically, we multiply the error by the **Transpose** of the weight matrix (U^T).

Scalar View: Total weighted error for h_1.

$$\begin{aligned} \text{Error from "I"}: &\quad 0.26 \times 0.2 = 0.052 \\ \text{Error from "love"}: &\quad 0.33 \times 0.4 = 0.132 \\ \text{Error from "NLP"}: &\quad -0.59 \times 0.6 = -0.354 \\ \text{Total}: &\quad 0.052 + 0.132 - 0.354 = -0.17 \end{aligned}$$

Matrix View

$$U^T \cdot \delta_o = \begin{bmatrix} 0.2 & 0.4 & 0.6 \\ 0.1 & 0.3 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.26 \\ 0.33 \\ -0.59 \end{bmatrix} = \begin{bmatrix} -0.17 \\ -0.17 \end{bmatrix}$$

Part B: The Activation Derivative (The "Switch" Factor)

We pushed the signal forward through the Sigmoid activation. When going backward, we must multiply by the derivative of the Sigmoid function ($\sigma'(z)$).

Why the Derivative?

The derivative tells us the sensitivity of the neuron. If the neuron was saturated (near 0 or 1), the slope is flat, and we dampen the error signal because changing the input wouldn't have changed the output much anyway.

The derivative of Sigmoid is: $\sigma'(z) = h(1 - h)$.

Scalar View: For h_1 (0.54):
$\sigma'(z_1) = 0.54 \times (1 - 0.54) \approx 0.248$

Part C: Combining A and B

We combine using element-wise multiplication (Hadamard product, $\odot$).

Scalar View: Final error for h_1.
$\delta_{h1} = -0.17 \times 0.248 \approx -0.042$

Matrix View

$$\delta_h = (U^T \cdot \delta_o) \odot \sigma'(z)$$

$$\delta_h = \begin{bmatrix} -0.17 \\ -0.17 \end{bmatrix} \odot \begin{bmatrix} 0.248 \\ 0.245 \end{bmatrix} \approx \begin{bmatrix} -0.042 \\ -0.042 \end{bmatrix}$$

Step 4: Update Input Weights (W)

We use the Golden Rule again. Destination = Hidden layer (δ_h); Source = Input layer (x).

Scalar View: Update the weight connecting the first input to the first hidden neuron ($W_{1,1}$, currently 0.5). Gradient = Error at h_1 × Input from x_1
Gradient = $-0.042 \times 0.1 = -0.0042$

Update Rule:
$W_{\text{new}} = 0.5 - (0.1 \times -0.0042) = 0.50042$

Matrix View

$\nabla W = \delta_h \cdot x^T$.

$$\nabla W = \begin{bmatrix} -0.042 \\ -0.042 \end{bmatrix} \cdot \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix}$$

$$\nabla W = \begin{bmatrix} -0.0042 & -0.0084 & \dots \\ \dots & \dots & \dots \end{bmatrix}$$

We have completed one full training cycle! The weights are now slightly better configured to predict "NLP" after "I love".

6 The Modern Landscape: LLMs and Transfer Learning

The principles we just walked through are the foundation of modern **Large Language Models (LLMs)** like GPT-4 or Claude. The core difference is scale (billions of parameters and massive datasets) and architecture (the Transformer architecture, which is more sophisticated but relies on the same backpropagation mechanics).

6.1 Generative vs. Discriminative AI

Feature	Discriminative AI	Generative AI (GenAI)
Goal	Classify existing data.	Create new, original data.
Example Task	Is this email Spam or Ham?	Write an email about a topic.
How it Learns	Learns the boundaries between classes.	Learns the underlying patterns and distribution o
Models	Logistic Regression, SVMs	LLMs (GPT, PaLM), Image Generators

6.2 Transfer Learning: Working Smarter

Training an LLM from scratch is prohibitively expensive. **Transfer Learning** is the standard approach in modern NLP.

1. **Pre-training (The Foundation):** A massive model is trained on a generic task (like next-word prediction) using a huge dataset (the internet). The model learns grammar, facts, and reasoning.
2. **Fine-tuning (The Specialization):** Take the pre-trained model and train it slightly on a specific, smaller dataset (e.g., legal documents).

This allows organizations to achieve high performance with minimal data and computation.

6.3 Prompt Engineering and In-Context Learning

LLMs exhibit **In-Context Learning**, the ability to learn from the prompt itself without updating their weights (i.e., without backpropagation).

- **Zero-Shot Inference:** Performing a task without examples.
  - *Prompt:* `Translate "Hello" to French.`
- **Few-Shot Inference:** Providing a few examples in the prompt to guide the model.
  - *Prompt:* `Apple -> Fruit. Cow -> Mammal. Sparrow ->`
  - *Output:* `Bird.`