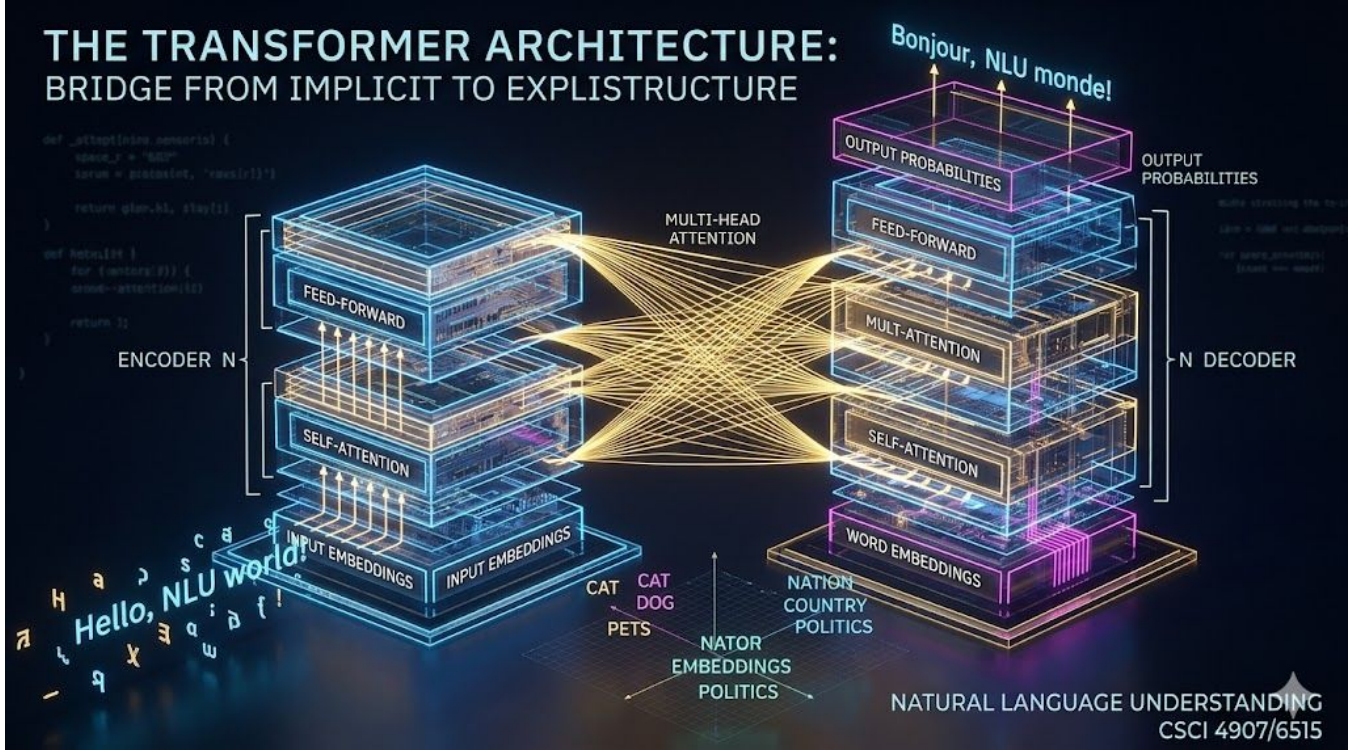


Natural Language Understanding

CSCI 4907/6515

Lecture 9: Introduction to Neural Networks

March 17, 2026

Aya Zirikly

The Biological Inspiration (1943)

The McCulloch-Pitts Neuron – Logical Foundations

Warren McCulloch and Walter Pitts proposed the first computational model of a "nerve net." They treated the neuron as a binary device (either firing/on or not/off).

The Mechanism:

- ❏ It takes multiple binary inputs (0 or 1)
- ❏ It applies a fixed **threshold**. If the sum of inputs exceeds the threshold, the output is 1

The Significance: This was the first time anyone proved that simple biological-like units could perform **universal logical operations** (AND, OR, NOT).

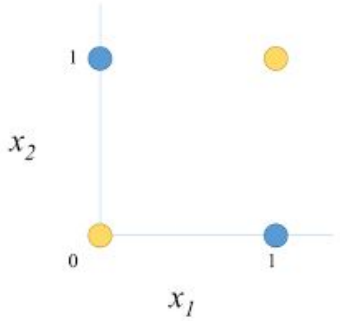
Limitation: There were no "weights" or "learning." The thresholds and connections had to be designed manually by humans, not learned from data.

The First Learning Machine (1958)

Rosenblatt’s Perceptron – The Dawn of Supervised Learning

- **The Breakthrough:** Frank Rosenblatt introduced **Weights** (w). Instead of just summing 1s and 0s, each input was multiplied by a weight that represented its importance.
- **The Perceptron Convergence Theorem:** Rosenblatt proved that if a solution existed (i.e., if the data was linearly separable), his algorithm was guaranteed to find it.
- **The First "NLU" Attempt:** Early Perceptrons were used for basic character recognition, attempting to "read" letters, a foundational task for NLU.

The "XOR" Wall and the First AI Winter



Linear Separability & The Minsky-Papert Critique (1969)

The Core Problem: Linear Separability

- Perceptrons are **linear classifiers**. This means they can only classify data that can be separated by a single straight line (in 2D) or a flat plane (in 3D).
- They work perfectly for "OR" and "AND" logic because you can draw a line between the "0" results and the "1" results.

The XOR (Exclusive OR) Trap

- XOR logic is different: The output is 1 ONLY if the inputs are different (1,0 or 0,1). If they are the same (0,0 or 1,1), the output is 0.
- If you plot these four points on a graph, you **cannot** draw one single straight line to separate the 0s from the 1s. They are "non-linearly separable."

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Why the AI Winter Happened

The Hardware Bottleneck

- 1960s computers were millions of times slower than a modern smartphone.
- Neural networks require massive matrix multiplications. Computers of that era (which often still used punch cards and had kilobytes of RAM) simply did not have the computational power to update thousands of weights.

The Data Drought

- Deep learning requires massive amounts of labeled data to adjust those weights accurately.
- In the 1960s and 70s, there was no internet, no digital text corpora, and no "Treebanks." The data required to train a complex NLU model simply did not exist.

The Deep Learning Renaissance

The Math is Solved: Backpropagation (1986)

- Researchers (Rumelhart, Hinton, Williams) popularized an algorithm that could efficiently calculate errors and update weights across *multiple* layers.
- Adding "Hidden Layers" finally allowed neural networks to solve non-linear problems like XOR.

The Machinery Catches Up: The GPU Era (2010s)

- The gaming industry accidentally saved AI. The Graphic Processing Units (GPUs) built to render video games turned out to be the exact hardware needed to process massive neural network matrices in parallel.

Big Data & The Web

- The rise of the internet provided limitless text data. We finally had the massive corpora (like Wikipedia and Common Crawl) needed to train complex Natural Language Understanding models.

Neural Networks vs. Logistic Regression

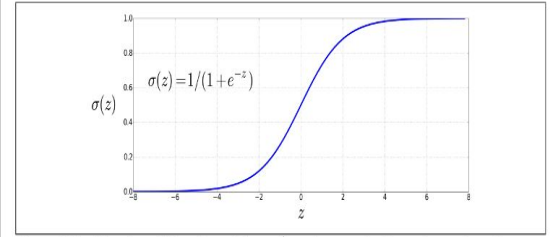


Figure 5.1 The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $(0, 1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

What actually IS a modern Artificial Neuron?

- Despite the biological name and the complex history, a single artificial neuron is mathematically almost identical to a model you already know: **Logistic Regression**.

The Mathematical Equivalence:

- **Step 1 (The Linear Part):** Both models take your inputs (features), multiply them by learned weights, and add a bias

  $z = w1x1 + w2x2 + ... + b$

- **Step 2 (The Non-Linear Part):** In Logistic Regression, you pass 'z' through a **Sigmoid function** to get a probability (output between 0 and 1)
  - In a Neural Network, you pass 'z' through an **Activation Function** (which can be Sigmoid, ReLU, or Tanh).

Input Case (x1, x2)	Target (OR Rule)	Calculation: z = w1(x1) + w2(x2) + b	Result of z	Sigmoid Probability	Threshold Check	Predicted Output
Case 1: (0, 0)	0	1(0) + 1(0) - 0.5	-0.5	~0.38	0.38 < 0.5	**0**
Case 2: (0, 1)	1	1(0) + 1(1) - 0.5	0.5	~0.62	0.62 > 0.5	**1**
Case 3: (1, 0)	1	1(1) + 1(0) - 0.5	0.5	~0.62	0.62 > 0.5	**1**
Case 4: (1, 1)	1	1(1) + 1(1) - 0.5	1.5	~0.82	0.82 > 0.5	**1**

Logistic regression is just ***one* neuron** → still a linear classifier (stuck at the XOR wall).

- Can only handle classes that are linearly separable

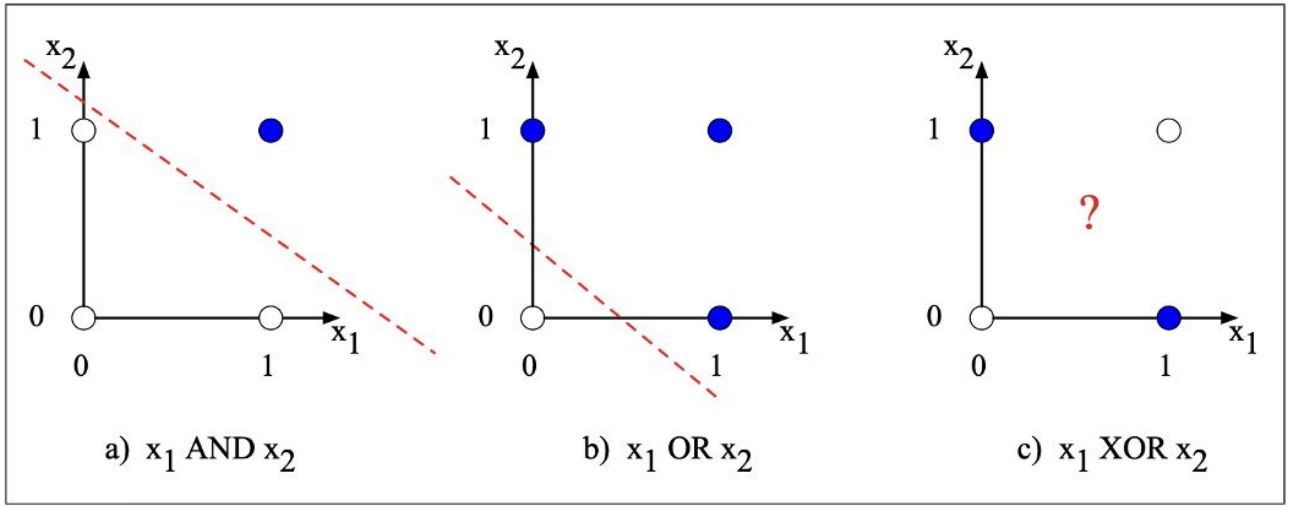


Figure 7.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y-axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

Logistic regression is just ***one* neuron** → still a linear classifier (stuck at the XOR wall).

- Can only handle classes that are linearly separable

Deep Learning → stack thousands of these "logistic regression" units together to learn complex, non-linear language patterns.

Single computational unit

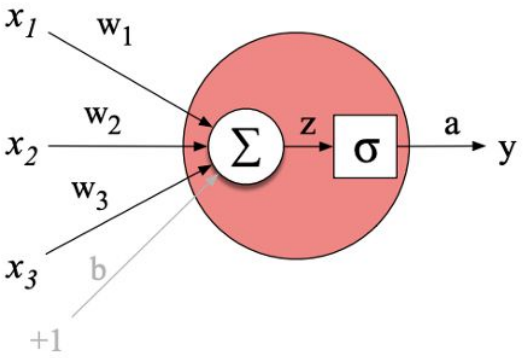


Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we rep
weight for an input clamped at +1) and producing an output y. We include some
intermediate variables: the output of the summation, z, and the output of the sig
this case the output of the unit y is the same as a, but in deeper networks we'll r
mean the final output of the entire network, leaving a as the activation of an indivi

Each neural unit multiplies input values by a weight vector, adds a bias, and then applies a non-linear activation function like sigmoid, tanh, or rectified linear unit (ReLU)

Activation functions

- Functions that are applied to the weighted sum to give unit’s output
- **The Non-Linear Step (Activation):** The neuron passes 'z' through an Activation Function to decide what its final output signal should be.
- Activation functions bend and curve the output, allowing the network to draw complex, non-linear boundaries (which is exactly what we need to solve the XOR problem and understand complex language).
- Sigmoid, softmax, tanh, ReLU

Sigmoid Function

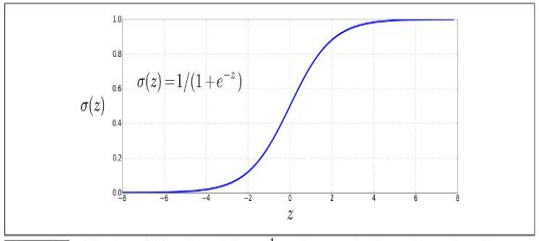


Figure 5.1 The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $(0, 1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

f(x) = 1 / (1 + e^{-x}) Outputs a value strictly between 0 and 1

Perfect for Probabilities: Because it forces all outputs between 0 and 1, it is ideal for the final output layer in binary classification (e.g., "Is this email spam? 0.9 = 90% yes").

Smooth Gradients: The curve is perfectly smooth, making it easy to calculate derivatives for Backpropagation.

Cons (Why we rarely use it in hidden layers):

- **The Vanishing Gradient Problem:** If the input 'x' is very large or very small, the curve gets perfectly flat. The gradient (slope) becomes almost zero. When the slope is zero, the network stops learning.
- **Not Zero-Centered:** The outputs are always positive, which can make the training process take zigzagging paths.

Sigmoid Function

The Zig-Zag Problem

- Because the inputs are strictly positive, the math forces the weight updates to move in the exact same direction.
- When the network tries to correct its errors, the weights are forced to either ALL increase or ALL decrease at the exact same time.
- Incapable of having some weights go up while others go down in the same step!

What is vanishing gradient problem

The Flat Tails: The curve at the two ends becomes almost completely horizontal.

What is the slope of a horizontal line? It is exactly **0** (or extremely close to 0, like 0.0001).

The Chain Reaction: Backpropagation works by multiplying gradients backwards through the layers (the Chain Rule).

- Imagine a deep network with 5 layers.
- If Layer 5 has a gradient of 0.1, and Layer 4 has a gradient of 0.1, the update for Layer 3 is `0.1 * 0.1 = 0.01`.
- If you keep multiplying small fractions together, the number shrinks exponentially.

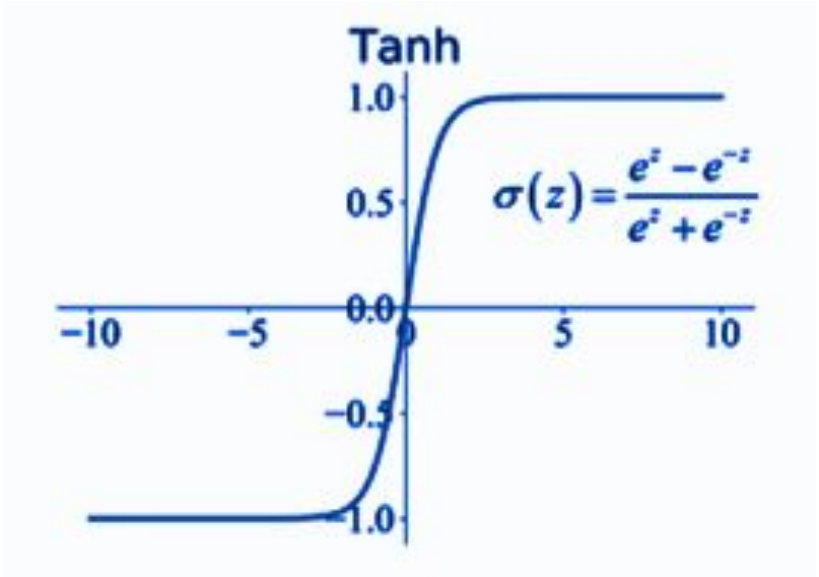
The Result (Vanishing): By the time the learning signal reaches the earliest layers of the network (Layer 1 or 2), the gradient has "vanished" to almost 0.00000001. Because the learning signal is effectively zero, the early layers **stop learning completely**

Network gets stuck.

The Tanh (Hyperbolic Tangent) Function

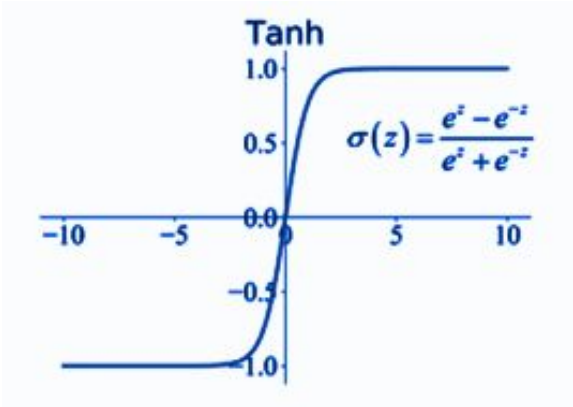
Zero-Centered Upgrade

Outputs between -1 and 1.



The Tanh (Hyperbolic Tangent) Function

- Shifted and stretched version of the Sigmoid function (It still has the classic "S-shape)."



Pros (Over Sigmoid):

- **Zero-Centered:** Because it ranges from -1 to +1, negative inputs yield negative outputs. This makes the math during backpropagation much cleaner and helps the model converge (learn) faster.
- Almost always preferred over Sigmoid for hidden layers in traditional Feed-Forward or Recurrent Neural Networks (RNNs).

Cons:

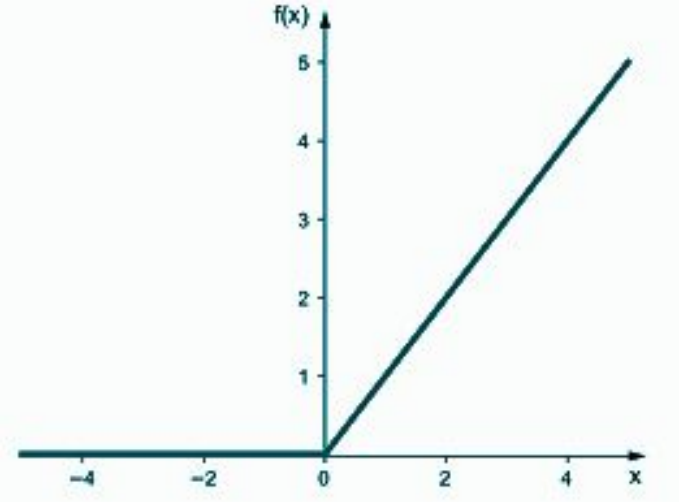
- **Still Vanishes**

Rectified Linear Unit -ReLU Function

f(x) = max(0, x)

Range: 0 to Infinity

- If the input is negative, output 0. If positive, output the exact same number



ReLU Function

Pros:

- **Solves Vanishing Gradients (Mostly):** On the positive side, the slope is always exactly 1. It never flattens out, meaning the gradient never vanishes. T
- **Extremely Fast:** ?

ReLU Function

Pros:

- **Solves Vanishing Gradients (Mostly):** On the positive side, the slope is always exactly 1. It never flattens out, meaning the gradient never vanishes. T
- **Extremely Fast:** Calculating e^-x (used in Sigmoid/Tanh) is computationally expensive. Calculating max(0, x) takes almost zero computing power.
- **Sparse Activation:** It outputs true zero for negative numbers, turning off unnecessary neurons and making the network highly efficient.

Cons:

- **The "Dying ReLU" Problem:** If a large weight update pushes a neuron's weights so that its output is always negative, that neuron will forever output 0. The gradient is zero, and the neuron essentially "dies" and never recovers.

Single computational unit

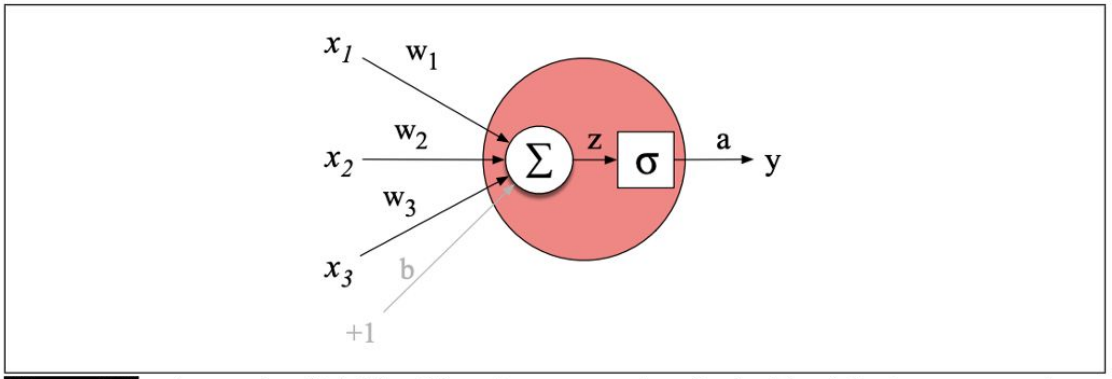


Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of the unit y is the same as a, but in deeper networks we'll reserve y to mean the final output of the entire network, leaving a as the activation of an individual node.

The power of neural networks comes from combining these & stacking them together

What is an Artificial Neural Network

- ❏ An Artificial Neural Network (ANN) is a computing system inspired by the biological neural networks that constitute animal brains.
- ❏ It is a machine learning framework that learns to perform tasks by considering examples, generally without being programmed with task-specific rules.

The Core Components:

- **Nodes (Neurons):** The basic units where computation happens.
- **Connections (Edges):** The pathways between neurons.
- **Weights:** Each connection has a "weight" that increases or decreases the strength of the signal. Learning is simply the process of adjusting these weights!
- **Biases:** An extra constant added to the neuron to shift the activation function, allowing the model to fit the data better.

Neural Networks in the real world

Machine Translation: Translating English to Japanese by mapping the structure of one language to another.

Sentiment Analysis & Hate Speech: Detecting the underlying emotion or toxicity in a block of text.

Generative AI: Writing code, generating poetry, or creating synthetic training data (like ChatGPT).

Non-text based:

- **Computer Vision:** Self-driving cars identifying pedestrians, or medical software detecting tumors in X-rays.
- **Speech Recognition:** Virtual assistants like Siri or Alexa converting audio waveforms into text.
- **Recommendation Systems:** Netflix or TikTok predicting what you want to watch next based on millions of past interactions.

Feedforward Neural Network (FFNN)

- ❏ Feedforward Neural Network is the simplest and most foundational type of artificial neural network.
- ❏ Information moves in only **one direction**: **forward**.

Structure

1. **Input Layer:** Receives the raw data (e.g., word embeddings or pixel values).
2. **Hidden Layer(s):** the network applies weights, biases, and activation functions to extract non-linear patterns.
3. **Output Layer:** Produces the final prediction (e.g., a probability score between 0 and 1).

FFNN in an example

- **Step 1: Input "The"**
  - The network processes "The" through its hidden layers and makes a prediction.
  - Forward pass complete & network resets completely.
- **Step 2: Input "cat"**
  - The network processes "cat".
  - Network has no idea that the word "The" just happened a millisecond ago.
- **Step 3: Input "sat"**
  - The network processes "sat". Again, it has no memory of "The" or "cat".

Basic neural network: Feedforward Neural Net

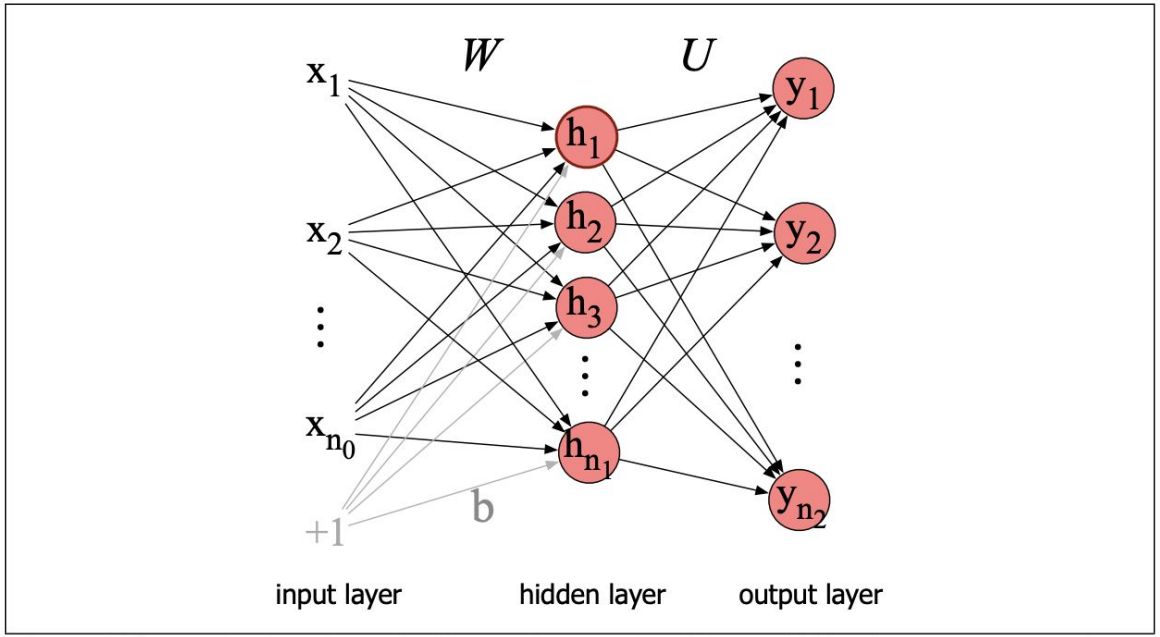


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Basic neural network: Feedforward Neural Net

(start by stipulating weights; then discuss learning them)

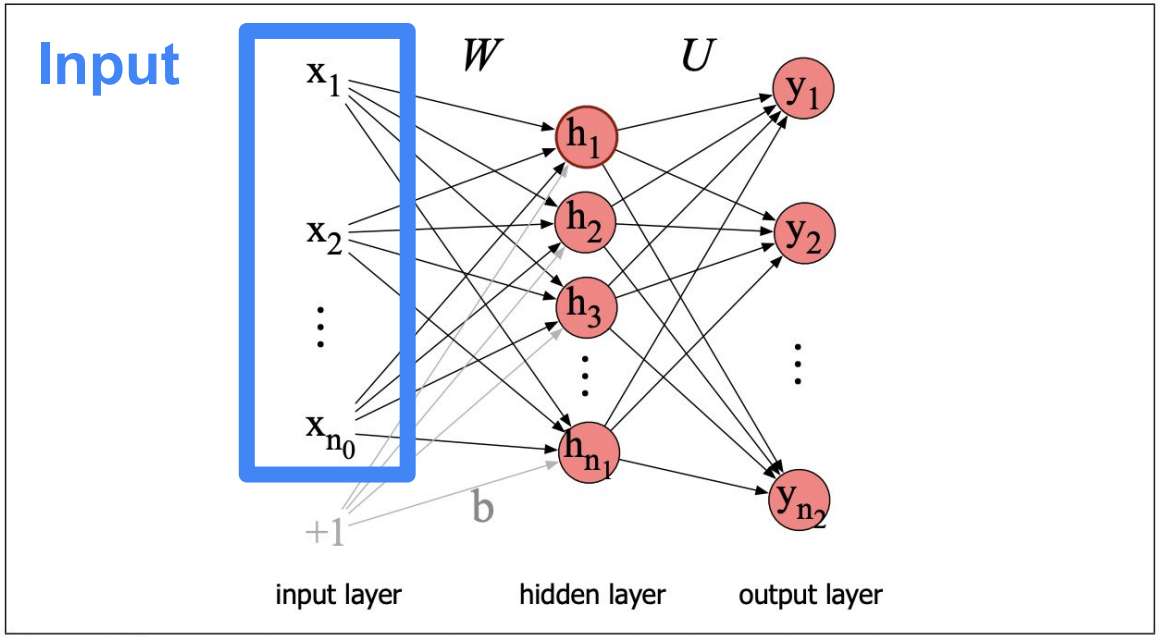


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Basic neural network: Feedforward Neural Net

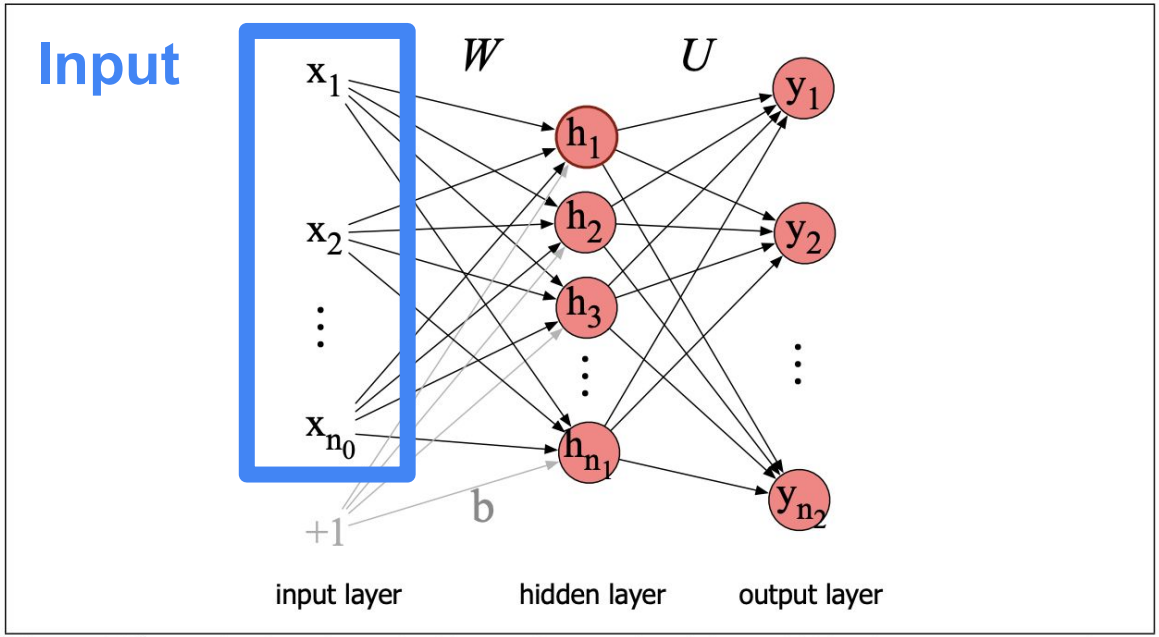


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Basic neural network: Feedforward Neural Net

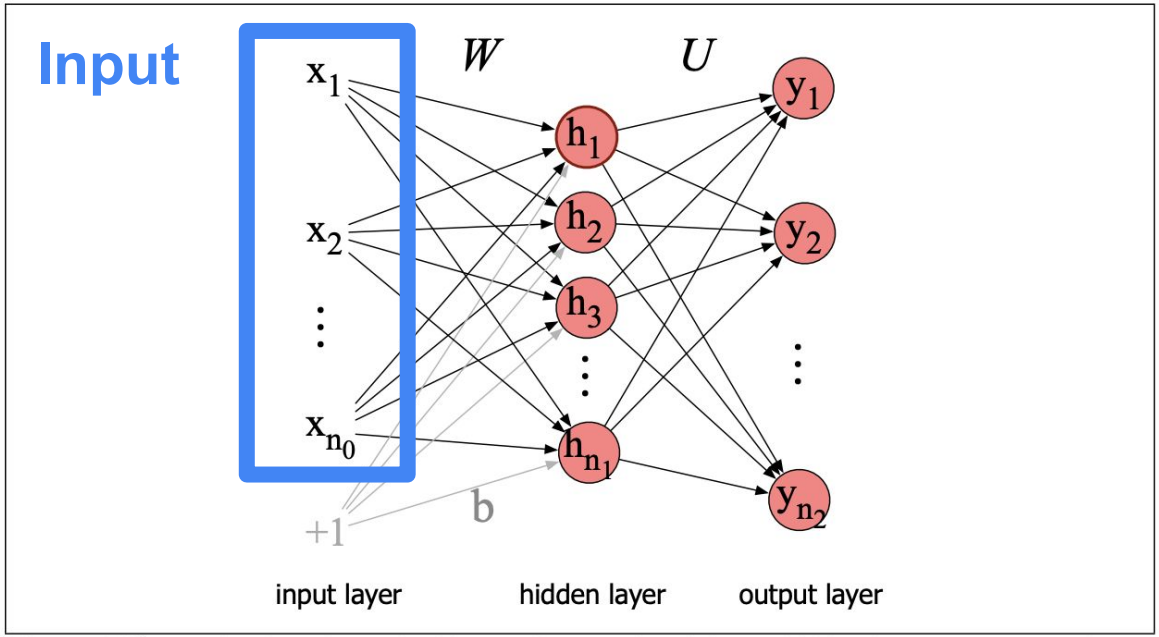


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Basic neural network: Feedforward Neural Net

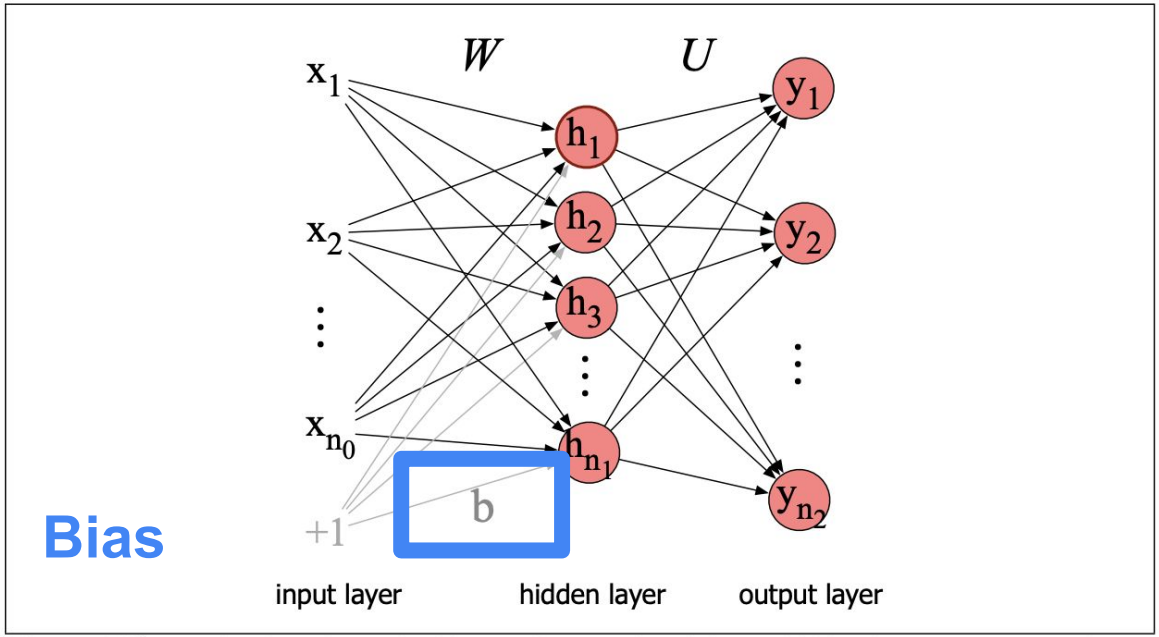


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Basic neural network: Feedforward Neural Net

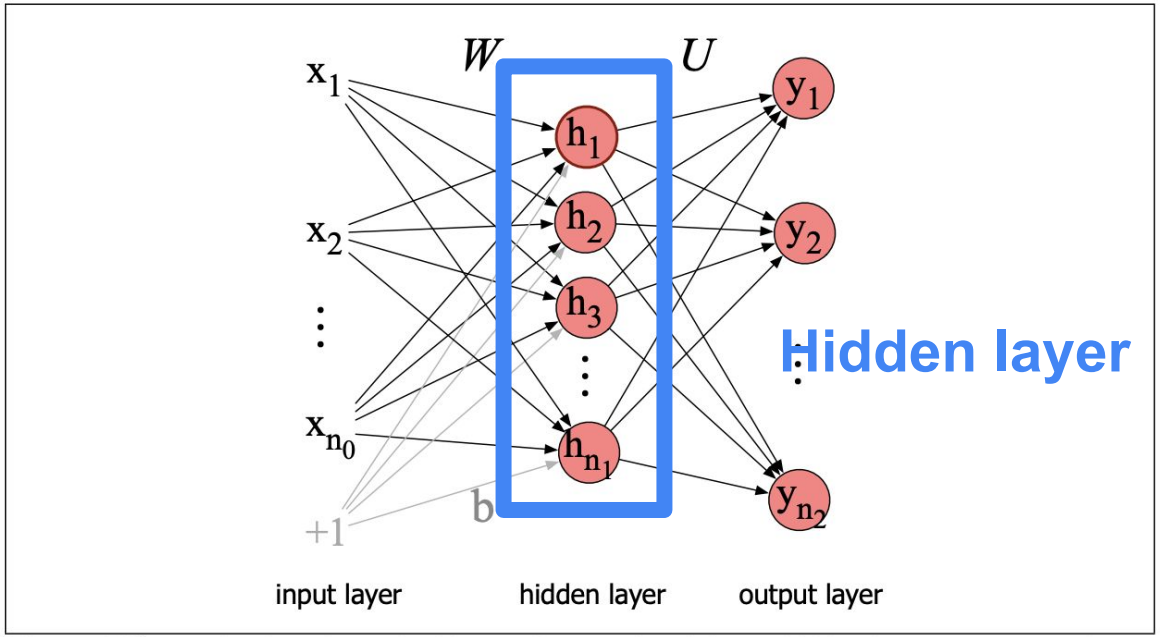


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Basic neural network: Feedforward Neural Net

Can have as many hidden layers as you want [GPT2 has 12; GPT3 has 96; GPT4 has 120

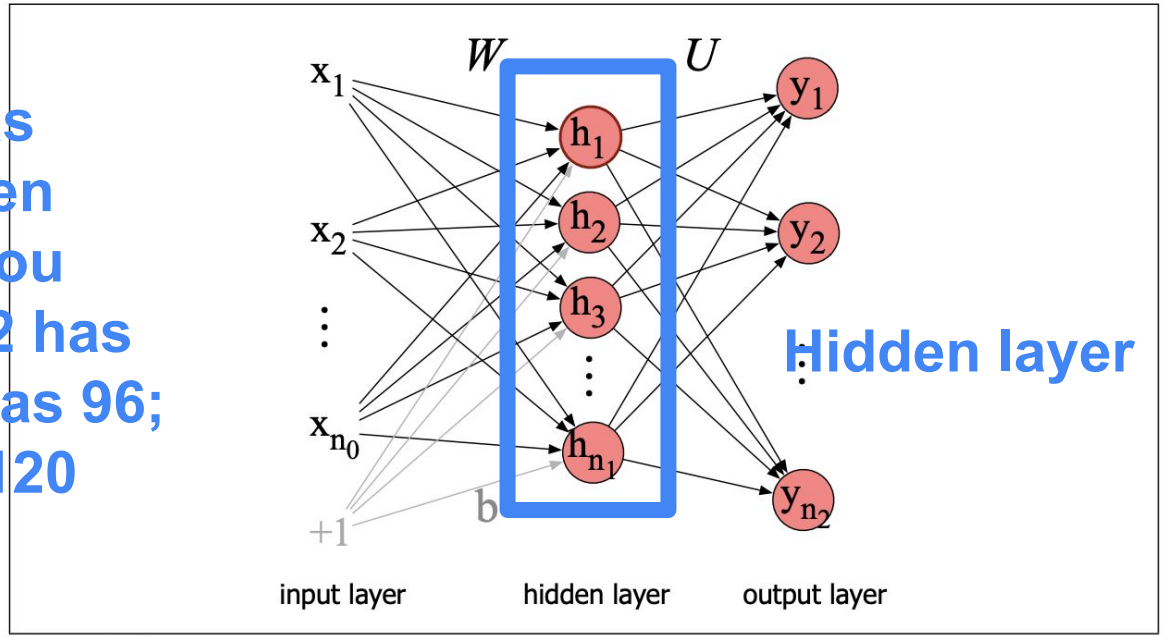


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Basic neural network: Feedforward Neural Net

Weights (w)

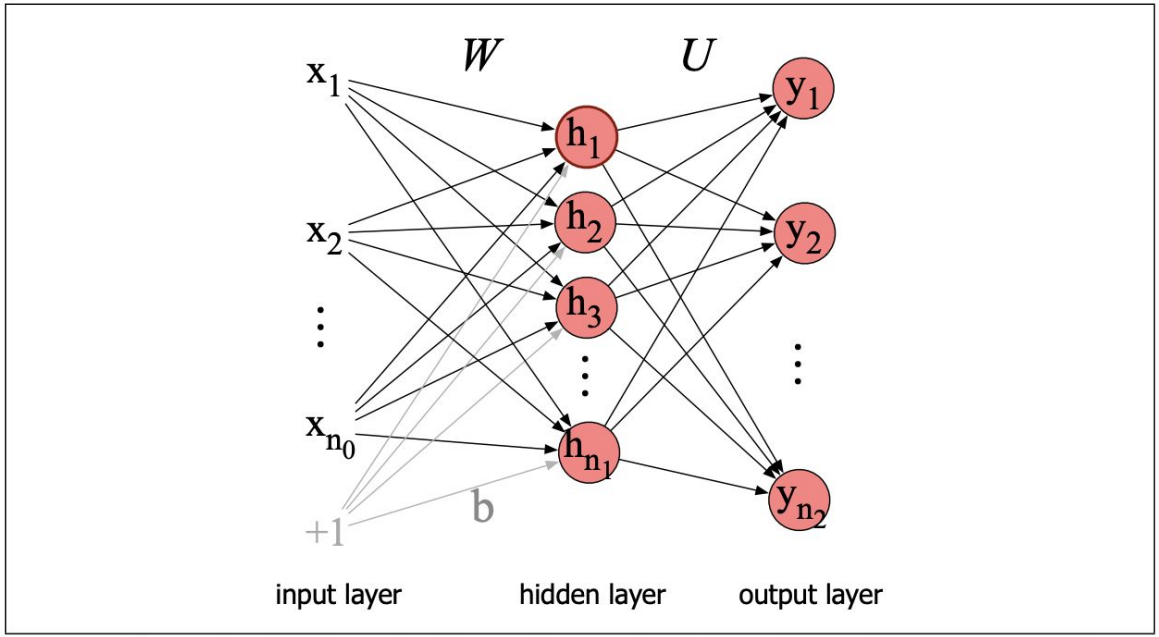
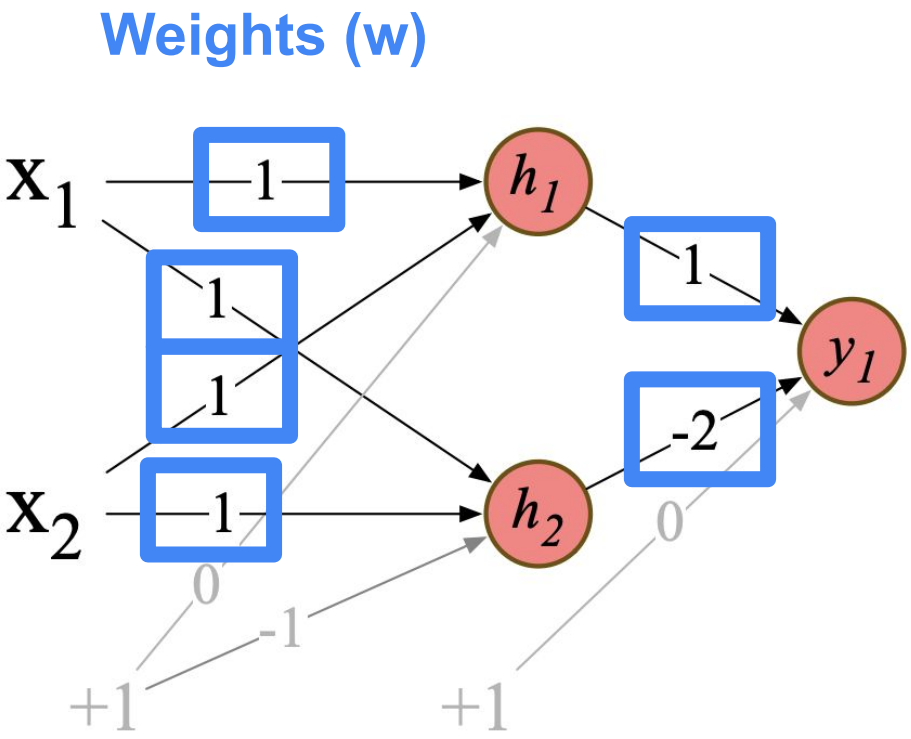


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Basic neural network: Feedforward Neural Net



Basic neural network: Feedforward Neural Net

Fully-connected = each unit in a layer takes as input the output from every unit in the previous layer; there is a link between all units in adjacent layers

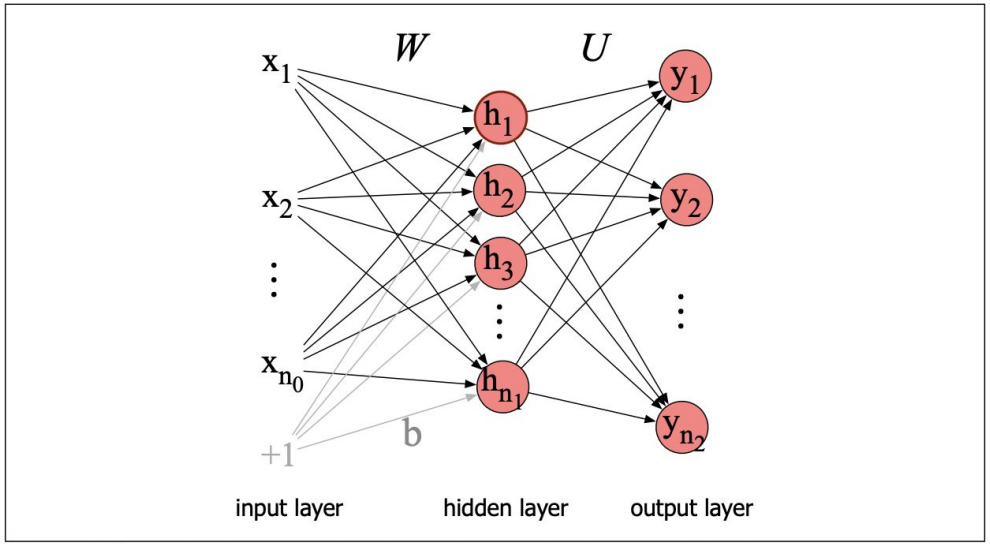


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Calculating feedforward output given a fully-specified neural network

1. Multiply inputs by weights (including bias term)
2. Feed the result into the activation function
3. Pass this output onto the next layer
4. Repeat, treating the output of the previous layer as the input to the current layer

Working example

The Architecture:

- 2 Input Nodes (x_1, x_2)
- 2 Hidden Nodes (h_1, h_2) using **ReLU** activation.
- 1 Output Node (y) using **Sigmoid** activation.
- **Inputs:** x_1 = 0.5, x_2 = 0.8
- **Hidden Layer Weights & Biases:**
    - Neuron h_1: Weights w_{11} = 0.2, w_{12} = -0.4, Bias b_1 = 0.1
    - Neuron h_2: Weights w_{21} = 0.6, w_{22} = 0.1, Bias b_2 = -0.2

Calculating Hidden Neuron 1 (h1):

1. **Linear Sum (z1):**
    - z1 = (0.5 * 0.2) + (0.8 * -0.4) + 0.1
    - z1 = 0.10 - 0.32 + 0.10
    - z1 = -0.12
2. **Activation (ReLU):** max(0, input)
    - h1 = max(0, -0.12)
    - **h1 = 0**

Calculating Hidden Neuron 2 (h2):

1. **Linear Sum (z2):**
    - z2 = (0.5 * 0.6) + (0.8 * 0.1) - 0.2
    - z2 = 0.30 + 0.08 - 0.20
    - z2 = 0.18
2. **Activation (ReLU):** max(0, input)
    - h2 = max(0, 0.18)
    - **h2 = 0.18**

(h1 = 0 and h2 = 0.18) are the new inputs for our final output neuron.

Calculating Output Neuron (y):

1. **Linear Sum (z_out):**
    - z_out = (h1 * v1) + (h2 * v2) + b_out
    - z_out = (0 * 0.5) + (0.18 * -0.3) + 0.2
    - z_out = 0 - 0.054 + 0.2
    - z_out = 0.146
2. **Activation (Sigmoid):** f(x) = 1 / (1 + e^-x)
    - y = 1 / (1 + e^-0.146)
    - y is approximately 1 / (1 + 0.864)
    - y is approximately 1 / 1.864
    - **Final Output: 0.536**

Loss

- Our network just predicted an output of **0.536** (53.6%).
- But to know if the network is smart or just guessing, we need the "Ground Truth" (the actual label from our training data).
- Let's say this specific example was a Hate Speech text, so the true Target label is **1** (100%).

Loss (Error):

- The goal of training is to figure out exactly how far off our prediction is from reality.
- We calculate a "Loss" score. A high loss means the network is doing terribly; a loss of 0 means the network is perfect.

Mean Squared Error vs. Cross-Entropy

Squared Error

- Formula:

  E = (Target - Prediction)^2

- E = (1 - 0.536)^2
- E = (0.464)^2
- **Loss = 0.215**

Cross Entropy

(when Target is 1): E = -log(Prediction) *(“log” here is the natural logarithm)*

E = -log(0.536)

Loss = 0.623

How a Neural Network actually learns from its mistakes

Backpropagation (The Feedback Loop)

- Backpropagation is the algorithm for determining the gradients, so as to do updates to the weights
- Once a network makes a prediction and calculates its Error (Loss), it needs to figure out which internal weights caused the mistake. Backpropagation is the process of sending that Error signal backward through the network to assign "*blame*" to each neuron.

The Fundamental Equation: The Chain Rule

To figure out how much a single, specific weight (w) is responsible for the final Error or Loss (L), we calculate the partial derivative (the rate of change) of the Loss with respect to that weight.

$$\frac{dL}{dw} = \frac{dL}{da} * \frac{da}{dz} * \frac{dz}{dw}$$

How a Neural Network actually learns from its mistakes

Learning Rate

Once we know who made the mistake and in what direction, the Learning Rate controls how much we change their behavior.

Neural Networks architecture design

- With neural networks, there are *many* decisions to make
- This is good in many ways: it’s a very flexible paradigm
- But - training is harder. You need to try a lot of things and see what works well.

Feedforward Neural Network (FFNN)

- ❏ Feedforward Neural Network is the simplest and most foundational type of artificial neural network.
- ❏ Information moves in only **one direction**: **forward**.

Structure

1. **Input Layer:** Receives the raw data (e.g., word embeddings or pixel values).
2. **Hidden Layer(s):** the network applies weights, biases, and activation functions to extract non-linear patterns.
3. **Output Layer:** Produces the final prediction (e.g., a probability score between 0 and 1).

Language is a Sequence, Not a Snapshot

The Limitation of FFNNs:

- Feedforward networks have **Amnesia**. They process each input completely independently.
- If you feed the network the word "Bank" in sentence A, and "Bank" in sentence B, it has no memory of the words that came before it to determine if it's a river bank or a financial institution.

The NLU Requirement:

- Human language is sequential. The meaning of a sentence depends entirely on the order of the words.
- To understand language, a neural network needs a concept of **Time** and **Memory**. It needs to remember what it just read a millisecond ago.

Recurrent Neural Network (RNN)

- A Recurrent Neural Network introduces a **Loop** (or recurrence) in its hidden layer.
- Instead of just passing information forward to the output, the hidden layer also passes its current state back to *itself* for the next step.

How it Works

- **Step 1:** The network reads the first word ("The"). It generates a "Hidden State" (a memory).
- **Step 2:** The network reads the second word ("cat"). It processes "cat" **PLUS** the Hidden State from Step 1.
- **Step 3:** The network reads the third word ("sat"). It processes "sat" **PLUS** the updated memory of "The cat".

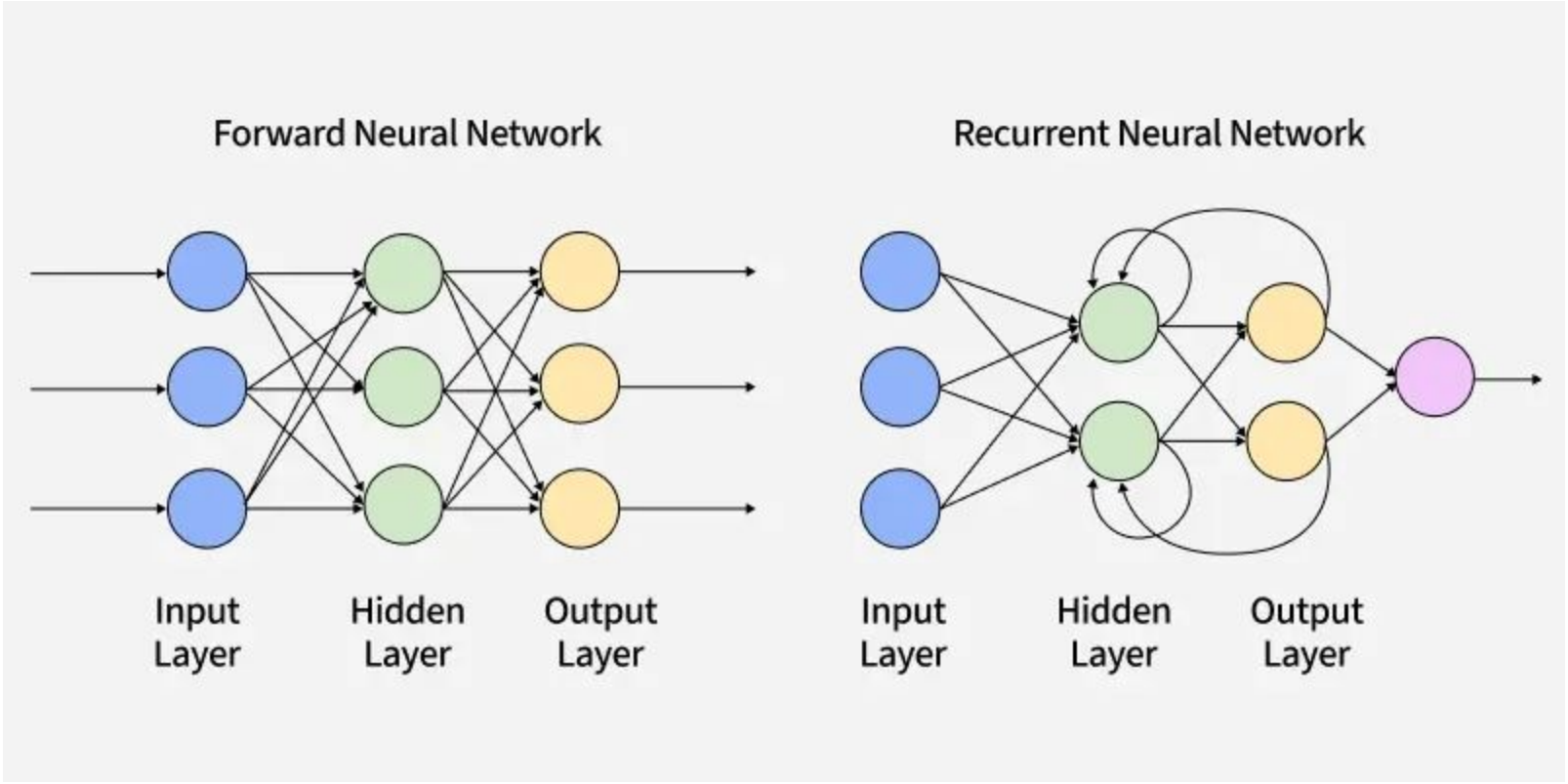
Recurrent Neural Network (RNN)

- A Recurrent Neural Network introduces a **Loop** (or recurrence) in its hidden layer.
- Instead of just passing information forward to the output, the hidden layer also passes its current state back to *itself* for the next step.

How it Works

- **Step 1:** The network reads the first word ("The"). It generates a "Hidden State" (a memory).
- **Step 2:** The network reads the second word ("cat"). It processes "cat" **PLUS** the Hidden State from Step 1.
- **Step 3:** The network reads the third word ("sat"). It processes "sat" **PLUS** the updated memory of "The cat".

The network builds a running context. The final output is influenced by the entire sequence of words, not just the current one.


Forward Neural Network
Input Layer
Hidden Layer
Output Layer
Recurrent Neural Network
Input Layer
Hidden Layer
Output Layer

Feature	Feedforward Neural Network (FFNN)	Recurrent Neural Network (RNN)
Data Type	Fixed-size, independent data (e.g., a single image).	Sequential, variable-length data (e.g., a sentence or paragraph).
Memory	**Stateless:** Has no memory of previous inputs.	**Stateful:** Maintains a "hidden state" across time steps.
Architecture	Information flows straight through (Input → Hidden → Output).	Contains loops (Hidden state loops back into itself).
Input Size	Must be exactly the same size every time.	Can handle sequences of any length (short phrases to long documents).
NLU Use Case	Basic classification (e.g., predicting the Part of Speech of one isolated word).	Sequence tasks (e.g., Machine Translation, Text Generation).

Classification using Neural Networks

Classification using neural networks

- The process of training a model to look at a piece of data (like an image, a sentence, or user behavior) and assign it to a specific, predefined label
- Instead of predicting a continuous number (like predicting tomorrow's temperature, which is called *regression*), a classification network predicts **probabilities**
  - How confident it is that the data belongs to class A, B, or C.

Classification using neural networks

The Input: You feed raw data into the network (e.g., the pixels of an image or the embedded words of a movie review).

The Hidden Layers: The network's hidden layers act as feature extractors. They find patterns such as edges and textures in an image, or sarcastic phrasing in a sentence.

The Output Layer: The final layer converts those learned patterns into a probability score for each possible label.

Types of classification

1. Binary Classification (Two Options)

- **The Goal:** Sorting data into exactly two categories (Yes/No, True/False, Class A/Class B).
- **The Output Layer:** Usually just **one single neuron** using the **Sigmoid** activation function.
- An email spam filter. The network outputs `0.95`, meaning it is 95% confident the email is "Spam" and 5% confident it is "Not Spam."

2. Multiclass Classification (Many Options, Choose One)

- **The Goal:** Sorting data into three or more categories, where the data can only belong to *one* category.
- **The Output Layer:** Has one neuron for *every* possible class, using the **Softmax** activation function (which forces all the output probabilities to add up to exactly 1.0 or 100%).
- A handwritten digit recognizer reading a zip code. It looks at a number and outputs: `[0: 5%, 1: 2%, 2: 90%, 3: 3%...]`. The model classifies it as a "2".

Types of classification

3. Multi-label Classification (Many Options, Choose Many)

- **The Goal:** Sorting data into multiple categories at the same time. The classes are not mutually exclusive.
- **The Output Layer:** Has multiple neurons, but uses **Sigmoid** on all of them independently, so they don't have to add up to 100%.
- Tagging a movie's genres. A single film can be simultaneously classified as `Action (98%)`, `Sci-Fi (85%)`, and `Comedy (12%)`.

Activation functions -revisited/summary

Sigmoid: The Independent Grader

- **The Formula:** f(x) = 1 / (1 + e^-x)
- **How it works:** It looks at each output node completely independently and squashes its value between 0 and 1.
- **The Rule:** If you have multiple Sigmoid nodes, their probabilities **do not** have to add up to 100%.
- **When to use it:** Binary Classification (Is this email Spam or Not Spam?)

Softmax: The Competitive Grader

- **How it works:** It looks at ALL the output nodes at the same time. It mathematically forces them to compete against each other.
- **The Rule:** It forces the sum of all output probabilities to equal exactly **1.0**
- **When to use it:** Multi-class Classification. When the input can only be exactly one thing (e.g., Is this handwritten digit a 0, 1, 2, 3... or 9?).

Input: we could use hand-engineered features…

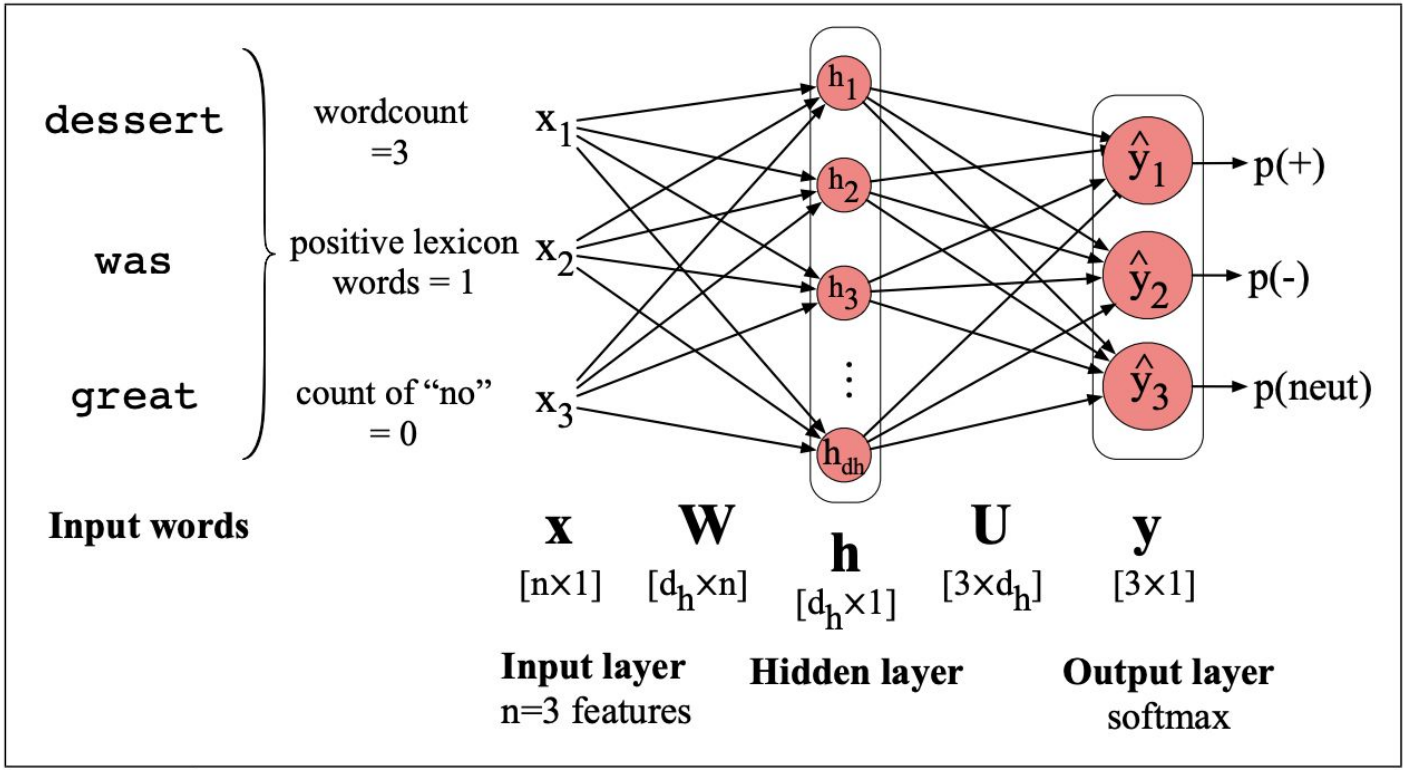


Figure 7.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

Input: but could also use embeddings!

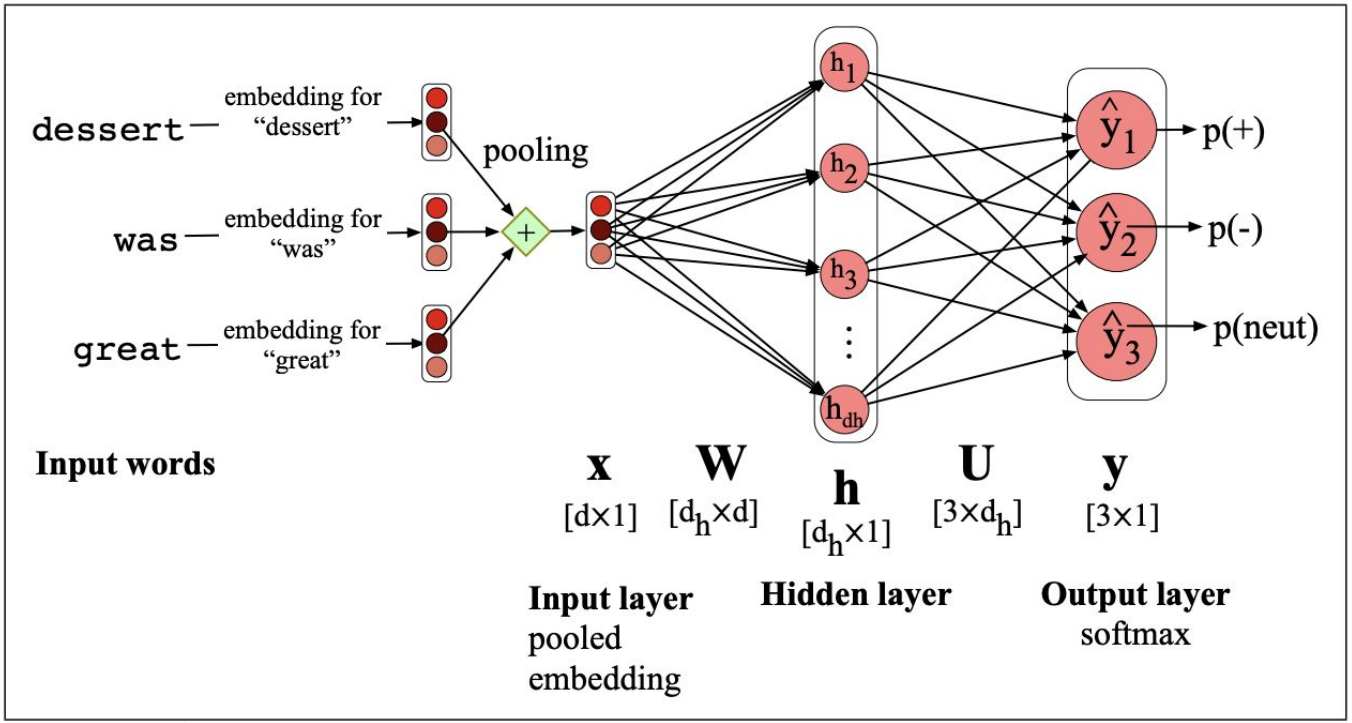


Figure 7.11 Feedforward network sentiment analysis using a pooled embedding of the input words.

Classifying using neural networks

Can think of hidden layers of neural nets as automatically learning features that help with classification task (rather than hand-selecting which features to input)

Benefits of embeddings over hand-engineered:

- can automatically learn the features that give best performance
- doesn’t have to be something a human could come up with

Downsides:

- not interpretable: hard to know what the features it is using are

Neural Networks as Automatic Feature Extractors

With deep learning, we completely skip the manual feature engineering step. We just feed the network raw data

Learned Representations:

- As the network trains using **Backpropagation**, the early hidden layers naturally learn to compress the raw data into dense arrays of numbers (embeddings) that capture the most important patterns.
- The network might learn to look for curves in layer 1, dog ears in layer 2, and a full dog face in layer 3—all without a human ever telling it what an "ear" is!

The Massive Benefits:

1. **Zero Human Intervention:** The network figures out what matters entirely on its own just by trying to minimize its Loss.
2. It discovers complex, non-linear relationships that humans could never manually code or even imagine.
3. **Task-Specific Optimization:** Because the embeddings are learned during training, they are custom-built to be the absolute perfect features for your specific classification task.

How this relates to backpropagation?

This is the true power of Backpropagation. When that Error signal travels backward, it isn't just fixing the final guess but redesigning the internal features (the embeddings) so the network makes a better prediction next time!"

- How might you build a neural network that predicts whether a title is a children's book title or adult title?

- How might you build a neural network that predicts the POS of a word?

- How might you build a neural network that predicts whether a title is a children's book title or adult title?

- How might you build a neural network that predicts the POS of a word?

Let's Build a Network: Children’s Book or Adult Book?

The Goal: We want to build a neural network that reads a book's title and predicts if it belongs in the Children's section or the Adult section.

Your Turn as the Architect!

Based on what we learned today, how would you design this system?

- **1. Data Prep:** How should we translate the words in the title into numbers for the network?
- **2. The Output Layer:** How many output nodes do we need, and what activation function should we use?
- **3. The Loss Function:** How should we calculate our error during training?

How We Build It

1. Data Prep (Learned Embeddings): Instead of using massive One-Hot arrays, we will use **Word Embeddings**. This allows the network to learn that words like "Magic" and "Puppy" cluster toward children's books, while "Tax" and "Murder" cluster toward adult books!

How We Build It

1. Data Prep (Learned Embeddings): Instead of using massive One-Hot arrays, we will use **Word Embeddings**. This allows the network to learn that words like "Magic" and "Puppy" cluster toward children's books, while "Tax" and "Murder" cluster toward adult books!

2. The Output Layer (1 Node + Sigmoid): Because this is a **Binary Classification** task (Children vs. Adult), we only need a single output neuron using the **Sigmoid** activation function to give us a probability from 0 to 1.

3. The Loss Function (Binary Cross-Entropy): Because we are using Sigmoid, we must use **Cross-Entropy** to calculate our error so the network doesn't suffer from the vanishing gradient problem when it makes a confidently wrong guess.

How to make architecture decisions

Don't Reinvent the Wheel (Follow the Literature)

- You almost never design an architecture entirely from scratch. You look at what the AI community has already proven works for your data type!
- **Text?** Start with an RNN or Transformer.
- **Images?** Start with a Convolutional Neural Network (CNN).

Start simple (the baseline)

- Always build a tiny, basic network first (or even run a standard Logistic Regression).
- *Why?* If your simple model gets 85% accuracy in two minutes, you now have a baseline. You know your complex, 50-layer network needs to beat 85% to be worth the effort.

How to make architecture decisions

- For standard Feedforward networks, hidden layers typically shrink as they get closer to the output (e.g., 512 nodes → 128 nodes → 32 nodes → 1).
- Think of it like a funnel: you are distilling a massive amount of raw data down into one final, concentrated decision.

Trial, Error, and Validation

- Deep learning is an **empirical science**.
- You will build 5 different versions of your network, train them all, and simply pick the one that gets the lowest Loss on a set of hold-out Validation/Dev Data.

What Happens When Our Network is Too Simple or Too Complex?

Scenario A - very small network

Imagine we give our network only one single hidden neuron to learn everything about the English language.

The Result: The network panics and looks for the easiest possible rule. It decides: *"If the title has the word 'The' in it, it's an Adult Book. If not, it's a Children's Book."*

The Reality: It is going to guess wrong almost every single time. It didn't have enough "brainpower" to learn the real patterns.

What Happens When Our Network is Too Simple or Too Complex?

Scenario B - too large network (memorizer network)

- Imagine we give our network 5,000 hidden layers. It has massive, almost infinite brainpower.
- **The Result:** Instead of learning general rules, it literally memorizes our exact training data. It learns: *"If the exact title is 'The Very Hungry Caterpillar', it's a Children's Book. If it is 'Goodnight Moon', it is a Children's Book. Literally anything else in the universe is an Adult Book."*
- **The Reality:** It gets a 100% perfect score on the books we trained it on, but if we hand it a brand new children's book it has never seen before, it fails completely.

What Happens When Our Network is Too Simple or Too Complex?

We need a network right in the middle. One that learns *general concepts* (like the fact that words like "Puppy" and "Magic" usually mean a book is for kids).

Did We Build the Right Size?

Underfitting (The Network is Too Small)

- **The Concept:** The architecture is too simple. It doesn't have enough layers or neurons to capture the complex patterns in the data.
- **The Result:** It performs terribly on the training data *and* terribly on the new validation data.
- **The Fix:** Increase the size of the network. Make it wider (more nodes) or deeper (more layers).

Did We Build the Right Size?

Overfitting (The Network is Too Massive)

- **The Concept:** The network has so many parameters that it literally memorizes the training data, including all the random noise, instead of learning the actual underlying rules.
- **The Result:** It gets near-perfect scores on the training data, but absolutely fails on the unseen validation data. It cannot generalize to the real world!
- **The Fix:** Shrink the network, get more training data, or use techniques like "Dropout" (randomly turning off neurons during training so they can't memorize the data).

The Goal: A network complex enough to learn the real patterns, but constrained enough that it is forced to generalize to new data.

How many layers

There is no magical mathematical formula for the perfect architecture, but there are strong rules of thumb based on how neural networks learn.

1. The Baseline (1 to 2 Hidden Layers)

- For standard, structured data (like spreadsheets or simple classification), 1 or 2 hidden layers is usually all you need.
- Making the layer *wider* (more nodes) is often enough to capture the necessary patterns.

2. Going "Deep" (3+ Hidden Layers)

- Deep learning really shines when dealing with complex, unstructured data.
- **Why depth matters:** Each layer builds on the previous one to learn a *hierarchy* of features.
- *Example:* Layer 1 learns simple lines. Layer 2 combines lines into shapes. Layer 3 combines shapes into a dog's face.

How many layers

3. The Golden Rule of Architecture

- Always start small. If your model is **Underfitting** (failing to learn), add more layers or nodes. If it starts **Overfitting** (memorizing the data), stop! You've gone too far.

Systematic testing (grid search)

Since we can't mathematically calculate the perfect architecture in advance, how do we find the best combination of layers, nodes, and learning rates? We automate the guessing!

The Problem: Hyperparameters

- These are the settings *you* control before training starts (e.g., Learning Rate, Number of Epochs, Number of Layers, Nodes per Layer).

The Brute-Force Solution: Grid Search

- Instead of manually guessing one at a time, you give the computer a list of options for each setting.
- *Example:* Try Learning Rates of [0.01, 0.1] and Hidden Layers of [1, 2, 3].
- Build and train a separate neural network for **every single possible combination** (in this case, 2 x 3 = 6 different models).
- Output the model that got the best score on the Validation/Dev Data

Grid Search is incredibly thorough, but it grows exponentially. If you want to test 4 learning rates, 4 layer sizes, and 4 activation functions, you have to train – entire neural networks from scratch!

Grid Search is incredibly thorough, but it grows exponentially. If you want to test 4 learning rates, 4 layer sizes, and 4 activation functions, you have to train 64 entire neural networks from scratch!

Random search

If Grid Search takes weeks to run, how do developers and researchers actually tune their networks in the real world?

Random Search.

How it Works:

- Instead of giving the computer a rigid list of numbers to check, you give it a **range** (e.g., "Try any Learning Rate between 0.001 and 0.1").
- Picks a completely random combination of settings, builds the network, trains it, and records the Validation score.
- **The Budget:** You tell the computer exactly when to stop. *"Try exactly 50 random combinations, and then just hand me the best one you found."*

Other parameters

- One single Epoch means the neural network has looked at your **entire training dataset** exactly one time.
- If you set Epochs = 50, the network will read through your entire dataset 50 separate times.

Finding the Sweet Spot:

- **Too Few Epochs (Underfitting):** Imagine reading a textbook only once before a final exam. You will fail. The network hasn't had enough time to adjust its weights.
- **Too Many Epochs (Overfitting):** Imagine reading the same textbook 10,000 times. You aren't learning the concepts anymore; you are just memorizing the page numbers. The network starts memorizing the training noise and fails on new data!

Batch size

If your dataset has 100,000 images, the network doesn't look at all 100,000 and *then* update its weights. That would take way too much computer memory! Instead, it takes "bites = batches"

What is Batch Size?

- The number of data points the network processes before it pauses, calculates the Loss, runs Backpropagation, and updates its weights.

The Three Approaches:

1. **Size = 1 (Stochastic):** The network updates its weights after every single image. It is very fast, but very chaotic. It might overreact to one weirdly drawn picture.
2. **Size = All Data (Full Batch):** The network calculates the perfect average error of the whole dataset before updating. It is very stable, but requires massive memory and can easily get stuck.
3. **Size = 32 or 64 (Mini-Batch):** The industry standard! It looks at a small chunk of data, gets a pretty good estimate of the error, and updates. It is the perfect balance of speed and stability.

Batch size

If your dataset has 100,000 images, the network doesn't look at all 100,000 and *then* update its weights. That would take way too much computer memory! Instead, it takes "bites = batches"

What is Batch Size?

- The number of data points the network processes before it pauses, calculates the Loss, runs Backpropagation, and updates its weights.

The Three Approaches:

1. **Size = 1 (Stochastic):** The network updates its weights aft[illegible] but very chaotic. It might overreact to one weirdly drawn picture.
2. **Size = All Data (Full Batch):** The network calculates the p[illegible]set before updating. It is very stable, but requires massive memory a[illegible]
3. **Size = 32 or 64 (Mini-Batch):** The industry standard! It looks at a small chunk of data, gets a pretty good estimate of the error, and updates. It is the perfect balance of speed and stability.

GPT4 ~60million tokens

Learning rate

- The size of the "step" the network takes when updating its weights to fix errors.
- Usually a tiny decimal (like 0.001).
  - If it's too big, the network becomes chaotic and never learns.
  - If it's too small, training takes weeks.

Evaluation metrics

Accuracy: The Overall Score

- **The Question:** Out of all the books, how many did we categorize correctly?
- (Total Correct Guesses) / (Total Guesses)
- Imagine our library has 99 Adult Books and only 1 Children's Book. If our lazy network just guesses "Adult Book" every single time, it gets a 99% Accuracy

Evaluation metrics

Accuracy: The Overall Score

- **The Question:** Out of all the books, how many did we categorize correctly?
- (Total Correct Guesses) / (Total Guesses)
- Imagine our library has 99 Adult Books and only 1 Children's Book. If our lazy network just guesses "Adult Book" every single time, it gets a 99% Accuracy... but it is completely useless at finding children's books!

Precision

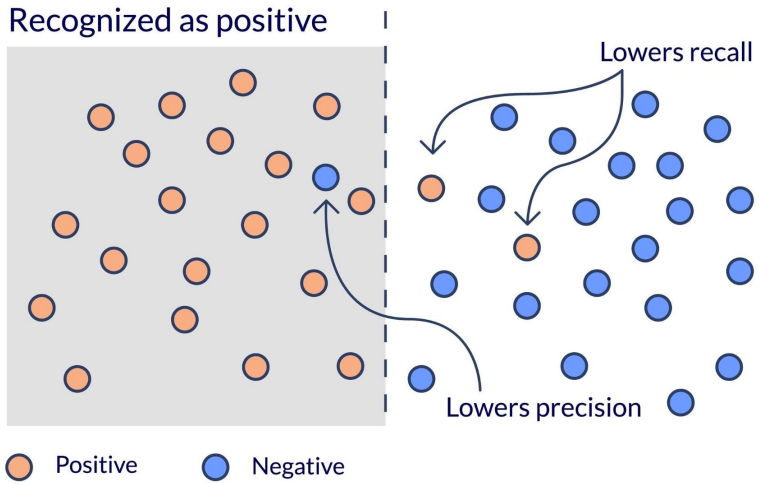
- **The Question:** When the network *claims* a book is for Children, how often is it actually correct?
- (Correct Children's Guesses) / (All Guesses that said "Children's")
- **Why we care:** If Precision is low, we have a "False Positive" problem.

Recall

- **The Question:** Out of all the *actual* Children's Books in the library, how many did our network manage to find?
- (Correct Children's Guesses) / (All Actual Children's Books)
- **Why we care:** If Recall is low, we have a "False Negative" problem.

If you want **High Precision** (never accidentally recommending a scary book to a kid)

- ❏ Your network will become overly cautious
- ❏ **Recall** will drop (it will miss some safe kids' books).
- ❏ You, the architect, have to decide which mistake is worse for your specific task/goal!



Language Modeling using Neural Nets

The Goal of Language Modeling: look at a sequence of words and mathematically guess which word comes next.

Building an FFNN language model

1. The Input (The Context Window): An FFNN has a fixed number of input nodes, so we have to tell it exactly how many previous words it is allowed to look at.

- Let's set our window to **3 words**. For the sentence *"The cat sat on the mat"*, our input is just the last 3 words: **["sat", "on", "the"]**.

2. The Hidden Layers (The Brain): We convert those 3 words into dense Word Embeddings (so the network knows "sat" is a verb and "the" is an article).

- The hidden layers multiply these embeddings together to extract the context (e.g., Ah, a location is coming next!).

3. The Output Layer (Softmax): The final layer has one node for *every single word in the English dictionary* (e.g., 50,000 nodes).

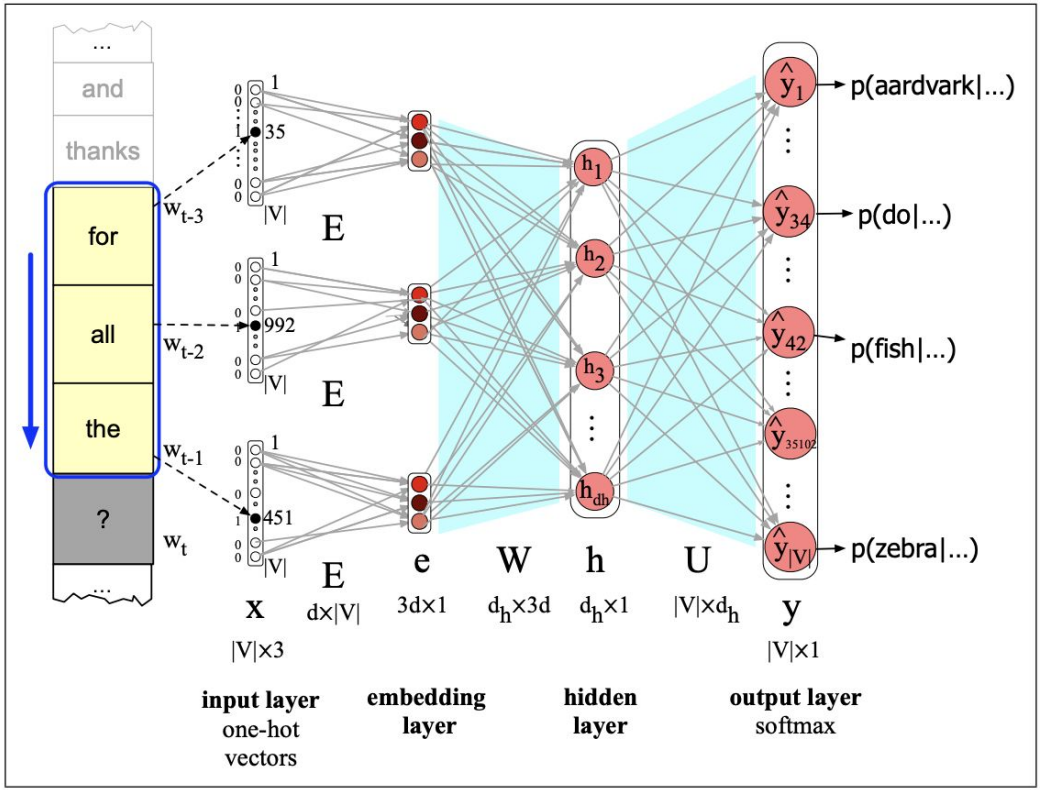
- We use the **Softmax** activation function to force all 50,000 nodes to compete, outputting a beautiful 0 to 1 probability for which word makes the most sense next!

If our context window is only 3 words, what happens if the sentence is: 'I grew up in France, so I speak fluent [French].'

If our context window is only 3 words, what happens if the sentence is: 'I grew up in France, so I speak fluent [French].'

The last 3 words are 'I, speak, fluent'. The network completely forgot about 'France' because it was outside the window!" This perfectly sets up the problem that Recurrent Neural Networks (RNNs) and Transformers were invented to solve!

Language modeling: How will we apply feedforward networks to predict the next word?



- Make independent predictions along the way
- Fixed size window of context words
- “Sliding window”

Making Independent Predictions

Because an FFNN has no memory of what it just did one second ago, how does it actually generate a long sentence?

➔ Looks at the exact words sitting in its input slots right now, runs the math, and spits out a probability. Every guess is a completely independent event.

The sliding window trick

To force the FFNN to write a continuous sentence, we have to manually shift the words in the input slots over by one space every time it guesses a new word [sliding window]

Let's watch it write with a 3-word window:

- **Step 1:** We feed it: ["The", "cat", "sat"] -> It independently calculates and predicts: **"on"**
- **Step 2:** We drop the oldest word ("The") and slide the new word ("on") into the window → We feed it: ["cat", "sat", "on"] → It independently calculates and predicts: **"the"**
- **Step 3:** We slide the window again → We feed it: ["sat", "on", "the"] → It independently calculates and predicts: **"mat"**

3. The Autoregressive Loop This process is called being "Autoregressive." The model consumes its own previous outputs as its new inputs. It will keep sliding and predicting indefinitely until it predicts a special "Stop" token!

Where else do we see autoregressive loop?

where a model consumes its own past outputs to make its next prediction

Modern Chatbots (GPT, Llama, DeepSeek)

- **How it's used:** Even though modern Large Language Models (LLMs) use advanced "Transformers" instead of standard FFNNs or RNNs, they are still fundamentally autoregressive!
- **The Loop:** When ChatGPT writes an essay, it is literally just predicting one single token (word piece) at a time, adding it to the screen, and then reading the whole screen again to guess the next token.

Where else do we see autoregressive loop?

where a model consumes its own past outputs to make its next prediction

Time Series Forecasting (Weather & Wall Street)

- **How it's used:** Long before neural networks, statisticians used AR (Autoregressive) models like **ARIMA** to predict continuous numbers over time.
- **The Loop:** To predict tomorrow's stock price or temperature, the model looks at the last 7 days. Once it guesses tomorrow's price, it adds that guess to the timeline so it can try to predict the day after tomorrow!

Training

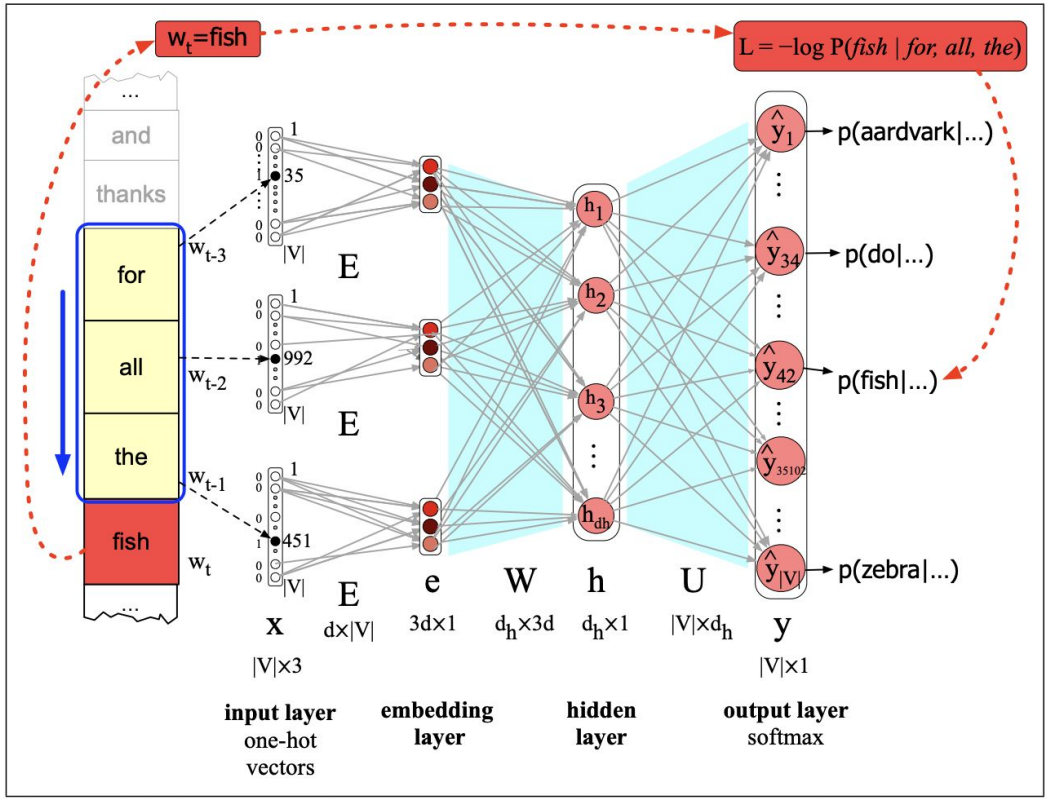


Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix **E** is shared among the 3 context words.

- Loss function
- Gradient descent
- Self-supervision (key concept in LLMs): train your model to predict the next word on real corpora; then you don't have to hand label anything

Language modeling using RNN

Instead of a fixed chunk of 3 words all at once, a Recurrent Neural Network (RNN) processes a sentence exactly like you do: sequentially, from left to right, carrying a running memory of what it just read.

How it works

I grew …

Step 1: The RNN reads the word "I". It creates a "Memory Vector" (called the Hidden State) that mathematically summarizes what it just read.

Step 2: It reads the next word, "grew". But instead of reading it in isolation, it mathematically combines "grew" with the **Memory Vector** from step 1. It updates its memory to mean: *"The subject is 'I', and the action is 'growing'."*

Step 3: It repeats this process for every word.

Stacked RNNs

How Stacking Works:

- **Layer 1 (The Grammar Layer):** The first RNN reads the raw Word Embeddings one by one. But instead of guessing the next word, it passes its "thoughts" up to the next layer. This layer might figure out basic syntax (e.g., *"This is a noun, that is a verb"*).
- **Layer 2 (The Meaning Layer):** The second RNN doesn't read the raw words; it reads the "thoughts" produced by Layer 1. It looks at the bigger picture to figure out semantics and tone (e.g., *"Ah, this sentence is meant to be a joke!"*).
- **The Output:** The very top RNN layer finally sends its highest-level thoughts to our trusty **Softmax** output layer, which calculates the final 0 to 100% probabilities for the next word.

Stacked RNNs

When to Stack: Use 2 to 4 layers when your task requires deep, abstract understanding of language.

- *Examples:* Machine Translation (translating English to Mandarin requires completely rethinking the grammar), generating long-form stories, or complex sentiment analysis.

When to STOP: Do *not* stack if your task is simple or your dataset is small.

- *Examples:* If you just want to classify whether a short movie review is "Positive" or "Negative," a single-layer RNN (or even an FFNN) is plenty.
- **The Trap:** Stacked RNNs take massively more time, memory, and computing power to train. If you stack 5 layers for a simple task, you are just wasting time and risking terrible Overfitting!

Stacked RNN cons

The Vanishing Gradient Problem

- Deep neural networks learn by passing error signals backwards (Backpropagation).
- In a Stacked RNN reading a long paragraph, that error signal has to travel back through time *and* down through multiple layers.
- **The Result:** The math gets multiplied by tiny fractions over and over. By the time the signal reaches the beginning of the sentence or the bottom layer, the signal is basically zero. The network literally stops learning!

Stacked RNN cons

The Memory Leak

- Because the signal vanishes, a Stacked RNN has severe short-term memory.
    a. When the network makes a mistake at the very end of a sentence, it has to send an error signal backwards to update the weights.
    b. In a normal network, it just goes back a few layers. But in a Stacked RNN, that signal has a massive commute: it has to travel backwards through **Time** (every single word) AND downwards through **Depth** (every stacked layer).
- It can translate a 5-word sentence perfectly. But if you give it a 50-word paragraph, by word 50, it has completely forgotten what word 1 was.

Stacked RNNs cons

We can't use standard RNNs for long documents. We need a new type of neuron—one that has a dedicated "Save" button and a "Delete" button so the math doesn't dilute over time.

Stacked RNNs

We can't use standard RNNs for long documents. We need a new type of neuron—one that has a dedicated "Save" button and a "Delete" button so the math doesn't dilute over time.

Long Short-Term Memory