9 Artificial Intelligence II

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

- Understand the basic architecture and key components of neural networks for regression and classification.
- Understand the architecture and principles of recurrent neural networks for processing sequential data.
- Understand the basic architecture and working principles of large language models (LLMs).

9.1 Neural Networks

This section discusses the key components of neural networks.

9.1.1 Perceptron

The perceptron is one of the earliest neural network models, composed of artificial neurons. Figure 9.1 illustrates a simple neuron with three inputs and one output. This model is conceptually similar to a biological neuron: input signals are first aggregated, and the result is then passed through a nonlinear activation function to produce an output.

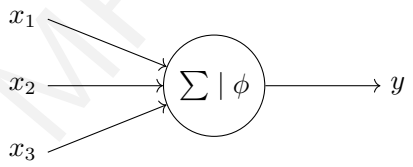


Figure 9.1: A simple perceptron with summation and activation function.

Mathematically, a perceptron computes a weighted sum of its inputs, followed by an activation function:

$$y = \phi(\mathbf{w}^\top \mathbf{x} + b),$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^\top$ is the input vector, $\mathbf{w} = (w_1, w_2, \ldots, w_n)^\top$ is the corresponding weight vector, b is a bias term, and $\phi(\cdot)$ denotes the activation function that introduces nonlinearity into the model.

9.1.2 Activation Functions

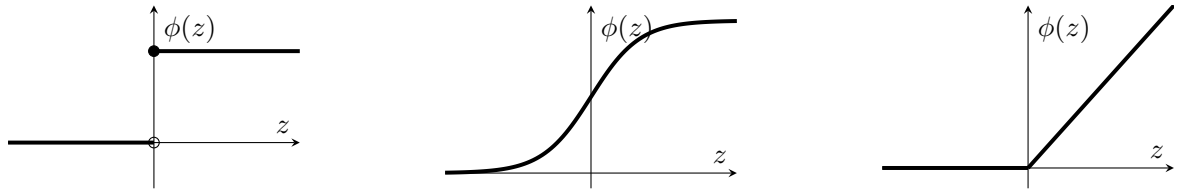
The activation function $\phi(z)$ introduces nonlinearity into the model by transforming the linear combination of inputs:

$$z = \mathbf{w}^\top \mathbf{x} + b, \qquad y = \phi(z).$$

Nonlinear activation functions enable neural networks to approximate complex, nonlinear relationships. Without an activation function, stacking multiple neurons would still result in a linear model.

MF 20006

There are several well-known activation functions, including the *step function*, the *sigmoid function*, and the *Rectified Linear Unit (ReLU)*. Different activation functions lead to different behaviors in terms of output range, smoothness, and optimization properties.



(a) Step function (b) Sigmoid function (c) ReLU function

Figure 9.2: Comparison of three activation functions: (a) step, (b) sigmoid, and (c) ReLU.

Step Function. The step function produces a binary output based on a fixed threshold:

$$\phi(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

It was originally used in the perceptron model to perform binary classification. However, because it is not differentiable, we cannot measure how changes in the input affect the output. As a result, this function is not suitable for gradient-based optimization methods.

Sigmoid Function. The sigmoid function is a smooth and differentiable approximation to the step function:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Its output lies strictly between 0 and 1, which allows it to be interpreted as a probability. Because of its smoothness, the sigmoid function enables gradient-based learning. However, for large values of $|z|$, the sigmoid function saturates, causing gradients to become very small. This phenomenon can slow down training in deep neural networks.

ReLU Function. The Rectified Linear Unit (ReLU) outputs zero for negative inputs and grows linearly for positive inputs. It is defined as:

$$\phi(z) = \max(0, z).$$

The gradient of ReLU is simple: 1 for $z > 0$ and 0 for $z < 0$. This property makes ReLU computationally efficient and helps alleviate the vanishing gradient problem, which is why it is widely used in modern deep neural networks.

9.1.3 Multi-layer Perceptron (MLP)

Although nonlinear activation functions can transform the output of a perceptron, a single-layer perceptron still produces a linear decision boundary for binary classification when using a monotonic activation function such as sigmoid. Let's consider a binary classification problem as shown in Figure 9.3, which classifies several one-dimensional points. Here, each input is directly mapped through the sigmoid function $y = \texttt{Sigmoid}(wx + b)$, and a threshold (usually $y = 0.5$) is used for classification. It is obvious that the two classes are not separable via the activation function.

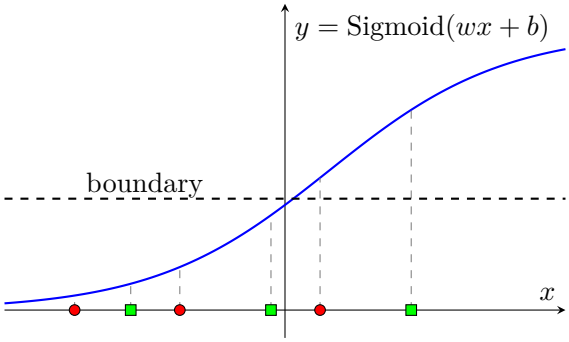


Figure 9.3: Mapping of one-dimensional inputs through the Sigmoid activation function.

The XOR problem is another classic example of a function with two inputs that cannot be linearly separated. Figure 9.4 shows the four possible input combinations for the XOR function, with the two output classes represented by different markers and colors.

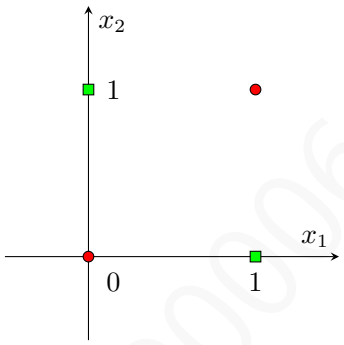


Figure 9.4: XOR problem: two-input binary classification.

To overcome the limitations of single-layer perceptrons, multiple perceptrons are stacked together to form a Multi-layer Perceptron (MLP). Figure 9.5 illustrates a simple MLP with three layers: an input layer, a hidden layer, and an output layer.

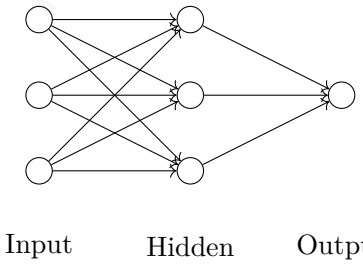


Figure 9.5: A simple MLP structure with one hidden layer.

In practice, MLPs can contain several hidden layers, each composed of multiple neurons. By applying nonlinear activation functions in the hidden layers, the network can compose complex nonlinear transformations, enabling it to model intricate relationships that a single-layer perceptron cannot capture. As an example, Figure 9.6 shows an MLP configured to solve the XOR problem, demonstrating how multiple layers allow the network to implement nonlinear decision boundaries. In this example, each neuron uses the step activation function, and the weights and biases are chosen to implement the XOR function.

Neural networks can naturally model regression, binary classification, and multi-class classification problems. For multi-class classification with C classes, the output layer contains C neurons, each producing

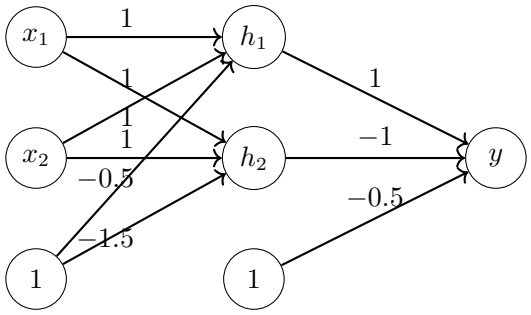


Figure 9.6: An MLP solution for the XOR problem.

a logit z_c for class c:

$$\mathbf{z} = [z_1, z_2, \ldots, z_C]^\top.$$

The logits are transformed into class probabilities using the softmax function, and the predicted class $\hat{c}$ is the one with the highest probability $\hat{y}_c$.

$$\hat{y}_c = \frac{\exp(z_c)}{\sum_{k=1}^{C} \exp(z_k)}, \quad c = 1, \ldots, C,$$

To train the network, we can compare the predicted probabilities $\hat{y}$ with the true class labels y using the cross-entropy loss:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{c=1}^{C} y_c \log \hat{y}_c,$$

where y_c is 1 if the true class is c and 0 otherwise (one-hot encoding).

MF 2000

9.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are neural networks designed to process sequential data, where the order of inputs matters. Examples include time-series data, such as daily temperatures and stock prices, and text sequences, such as sentences.

9.2.1 RNN Architecture

Consider an input sequence $x_1, x_2, \ldots, x_n$, where $x_i \in \mathcal{R}^d$ represents the input vector at time step i. At each time step, the RNN updates its hidden state using the current input and the previous hidden state:

$$h_i = \tanh(Wx_i + Uh_{i-1} + b_h),$$

where $h_i \in \mathcal{R}^m$ is the hidden state, $W \in \mathcal{R}^{m \times d}$ and $U \in \mathcal{R}^{m \times m}$ are trainable weight matrices, and b_h is a bias vector.

The output at each time step is computed as:

$$y_i = Vh_i + b_y,$$

where V is another trainable weight matrix. Depending on the task, an additional activation function, such as softmax, may be applied to the output.

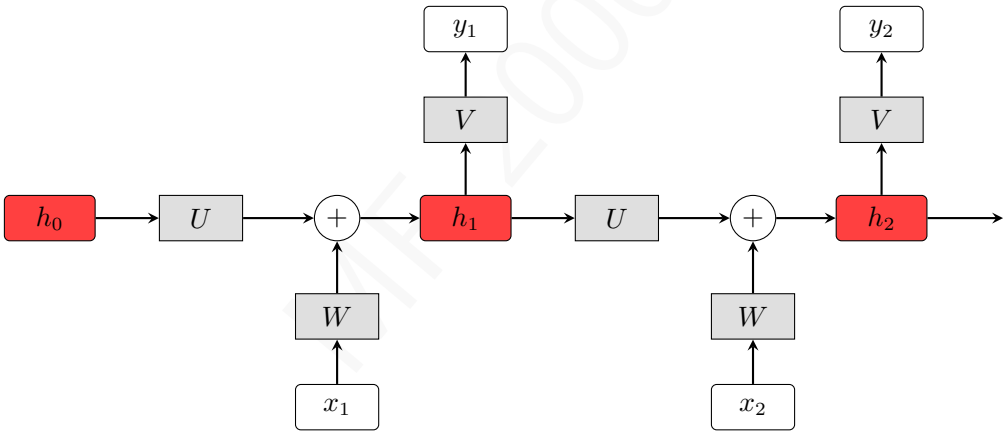


Figure 9.7: Architecture of a recurrent neural network unfolded over two time steps.

As illustrated in Figure 9.7, an RNN processes inputs sequentially. At each time step, the hidden state combines information from the current input and the previous hidden state. The weight matrices W, U, and V are shared across all time steps.

RNNs have been successfully applied to various sequential tasks, including time-series prediction and natural language processing. However, they suffer from vanishing gradients, which make it difficult to learn long-range dependencies, and sequential computation, which limits parallelism. Long Short-Term Memory (LSTM) networks were developed to better capture long-range dependencies. Despite these improvements, their performance remains limited, especially in complex language processing tasks. More recently, Transformer models have achieved significant advances through attention mechanisms and greater parallelism, forming the foundation of modern large language models.

9.3 Large Language Models

Large language models (LLMs) are neural networks designed to understand and generate natural language. Most modern LLMs are based on the Transformer architecture and generate text by predicting the next token based on the preceding context.

In this section, we introduce the key components of LLMs, including tokenization, word embeddings, attention mechanisms, and Transformer layers.

9.3.1 Word Embedding

The first step in language modeling is to convert natural language into a mathematical representation that can be processed computationally. Since neural networks operate on numerical inputs, words or symbols in natural language must be mapped into a vector space. This process of mapping discrete symbols to continuous vector representations is called word embedding. In word embedding, each token is represented by a d-dimensional real-valued vector in a shared vector space $\mathcal{R}^d$.

One of the simplest approaches to representing tokens is one-hot encoding. Given a vocabulary of size V, each token is represented by a V-dimensional vector containing a single 1 and zeros in all other positions. However, this representation has several limitations: it is high-dimensional and sparse, and it does not capture semantic relationships between tokens.

Word embedding addresses these limitations by mapping one-hot vectors into a lower-dimensional continuous space. Mathematically, word embedding can be viewed as a linear transformation of a one-hot vector. Let $W \in \mathcal{R}^{V \times d}$ be an embedding matrix. The embedding of token i can be expressed as:

$$e_i = o_i W$$

where o_i is the corresponding one-hot vector.

Since o_i contains a 1 only at position i, this operation is equivalent to selecting the i-th row of W. Therefore, in practice, one-hot vectors are not explicitly constructed. Instead, embeddings are obtained through a lookup operation.

The key properties of word embeddings are their dense representation and learnability. Unlike one-hot vectors, embeddings are typically lower-dimensional and dense. Their values are automatically learned during training, allowing the vector space to capture semantic and syntactic relationships between words.

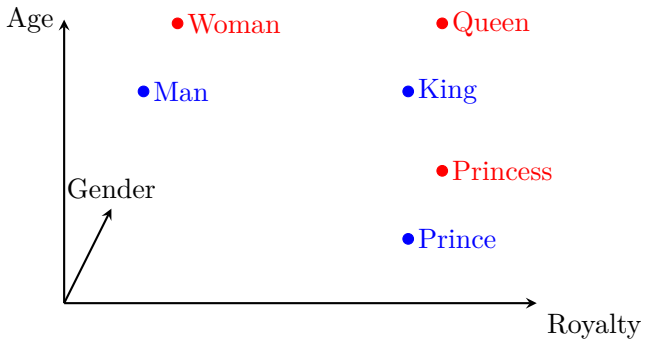


Figure 9.8: Illustration of word embeddings

If word embeddings are learned effectively, semantic and syntactic relationships between words can be reflected in the vector space. For example, the well-known analogy

$$\text{King} - \text{Man} + \text{Woman} \approx \text{Queen}$$

illustrates how gender relationships may be represented through vector arithmetic. Similarly,

$$\text{King} - \text{Prince} \approx \text{Queen} - \text{Princess}$$

illustrates how age-related relationships may be captured in embeddings of royal titles.

How can such a vector space be constructed, and how can discrete words or tokens be mapped into it? The central idea is to let the model automatically learn vector representations through a learning task, rather than manually defining the meaning of each vector dimension. In modern large language models, embeddings are generally not trained as a separate component. Instead, they are learned end-to-end as part of the entire model. Specifically, the embedding matrix W is treated as a set of trainable parameters and optimized jointly with the subsequent neural network layers.

9.3.2 Tokenization

In large language models, the basic units used for embedding are not necessarily complete words. A vocabulary may contain individual characters, complete words, or smaller units known as subwords. Modern language models commonly employ subword-based tokenization methods. One possible tokenization strategy is longest-prefix matching, which selects the longest token in the vocabulary that matches the current prefix of the input text.

For example, suppose the vocabulary contains:

`p`, `l`, `a`, `y`, `i`, `n`, `g`, `play`, `ing`

Given the input string:

`playing`

A longest-prefix-matching tokenizer would first select `play`, followed by `ing`, rather than processing the string character by character.

Subword tokenization provides several advantages with a vocabulary of limited size:

- Frequently occurring words can be represented as complete tokens.
- Rare words can be decomposed into smaller subword units.
- Out-of-vocabulary (OOV) words can often be represented using known subwords.
- The model can exhibit some robustness to spelling errors and unfamiliar word forms.

MF 20006

9.3.3 Attention Mechanism

Unlike RNNs, which process tokens sequentially through hidden states, attention allows each token to directly capture information from other tokens in a sequence. This enables more effective modeling of long-range dependencies and greater parallelism during training.

Attention is a key component of the Transformer architecture, which forms the foundation of modern large language models (LLMs). Its main idea is to compute relationships between tokens and aggregate information through weighted combinations of their representations.

To implement attention, each input token representation is projected into three vectors: Query (Q), Key (K), and Value (V). The Query and Key vectors are used to compute attention scores, which determine how much information each token receives from other tokens. These scores are then used to compute a weighted combination of the Value vectors. The standard scaled dot-product attention is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

where

$$Q \in \mathcal{R}^{n\times d_k}, \quad K \in \mathcal{R}^{n\times d_k}, \quad V \in \mathcal{R}^{n\times d_v}.$$

Here, n denotes the sequence length, d_k is the dimensionality of the Query and Key vectors, and d_v is the dimensionality of the Value vectors. The scaling factor $\sqrt{d_k}$ prevents the dot products from becoming excessively large as the dimensionality increases, helping maintain stable gradients through the softmax operation.

Consider an input sequence of token representations:

$$x_1, x_2, \ldots, x_n \in \mathcal{R}^{d_{\text{model}}},$$

where d_{model} denotes the dimensionality of the input representations. Before computing attention, each token is projected into Query, Key, and Value vectors through three learned linear transformations:

$$\begin{aligned} q_i &= x_i W_Q, \\ k_i &= x_i W_K, \\ v_i &= x_i W_V, \end{aligned}$$

where the projection matrices are:

$$W_Q \in \mathcal{R}^{d_{\text{model}}\times d_k}, \quad W_K \in \mathcal{R}^{d_{\text{model}}\times d_k}, \quad W_V \in \mathcal{R}^{d_{\text{model}}\times d_v}.$$

Therefore:

$$q_i \in \mathcal{R}^{d_k}, \quad k_i \in \mathcal{R}^{d_k}, \quad v_i \in \mathcal{R}^{d_v}.$$

Within a single attention head, Queries and Keys are projected into a shared feature space $\mathcal{R}^{d_k}$ for similarity computation, while Values are projected into $\mathcal{R}^{d_v}$ for information aggregation.

NF 2006

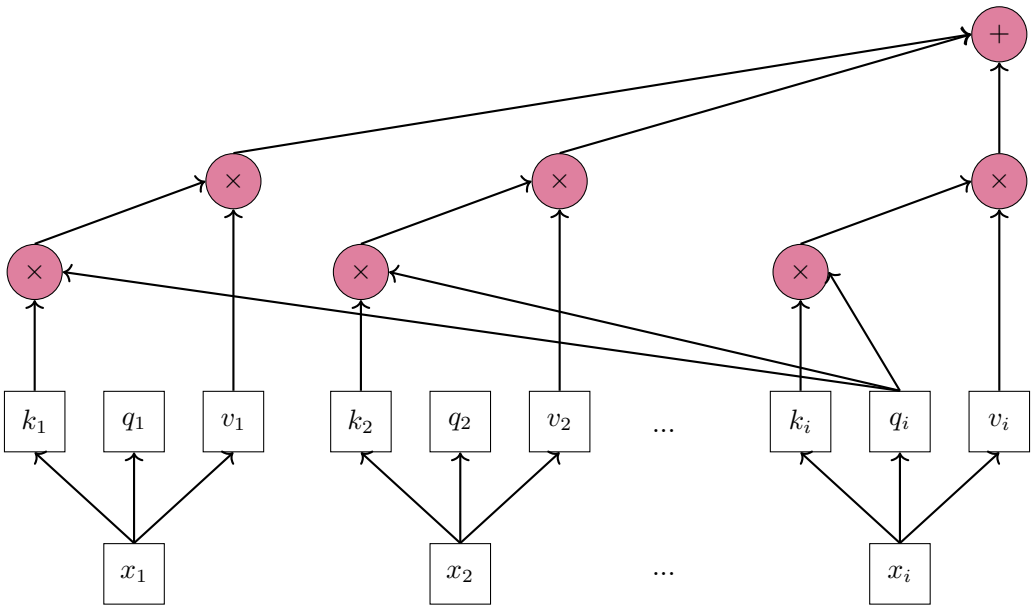


Figure 9.9: Attention computation for token x_i

Figure 9.9 illustrates the attention computation for token x_i, using its Query vector to attend to the preceding tokens. The computation consists of the following steps:

1) Consider an input sequence $x_1, x_2, \ldots, x_i \in \mathcal{R}^{d_{\text{model}}}$. The Query vector for token i is obtained through a linear transformation:

$$q_i = x_i W_Q.$$

For each token $j \leq i$, the corresponding Key and Value vectors are computed as:

$$k_j = x_j W_K, \quad v_j = x_j W_V,$$

where $q_i, k_j \in \mathcal{R}^{d_k}$ and $v_j \in \mathcal{R}^{d_v}$.

2) Compute the dot-product similarity between the Query and each Key:

$$s_{ij} = q_i \cdot k_j.$$

The colored $\times$ nodes in the figure represent these operations.

3) Normalize the scaled attention scores using softmax:

$$\alpha_{ij} = \frac{\exp(s_{ij}/\sqrt{d_k})}{\sum_{t=1}^{i} \exp(s_{it}/\sqrt{d_k})}, \quad \sum_{j=1}^{i} \alpha_{ij} = 1.$$

4) Compute the weighted sum of the Value vectors to obtain the attention output:

$$\text{out}_i = \sum_{j=1}^{i} \alpha_{ij} v_j \in \mathcal{R}^{d_v}.$$

The colored + node in the figure represents this aggregation operation.

MT2006

9.3.4 Decoder-only Transformer

The Transformer architecture builds on attention mechanisms to process sequences without recurrent connections. Modern LLMs commonly use a Decoder-only Transformer, which consists of multiple stacked Transformer blocks. Each block uses Masked Self-Attention to capture relationships between tokens while preventing access to future tokens. This allows the model to perform autoregressive language modeling, predicting each token based on the preceding context:

$$P(x_i \mid x_1, x_2, \ldots, x_{i-1}).$$

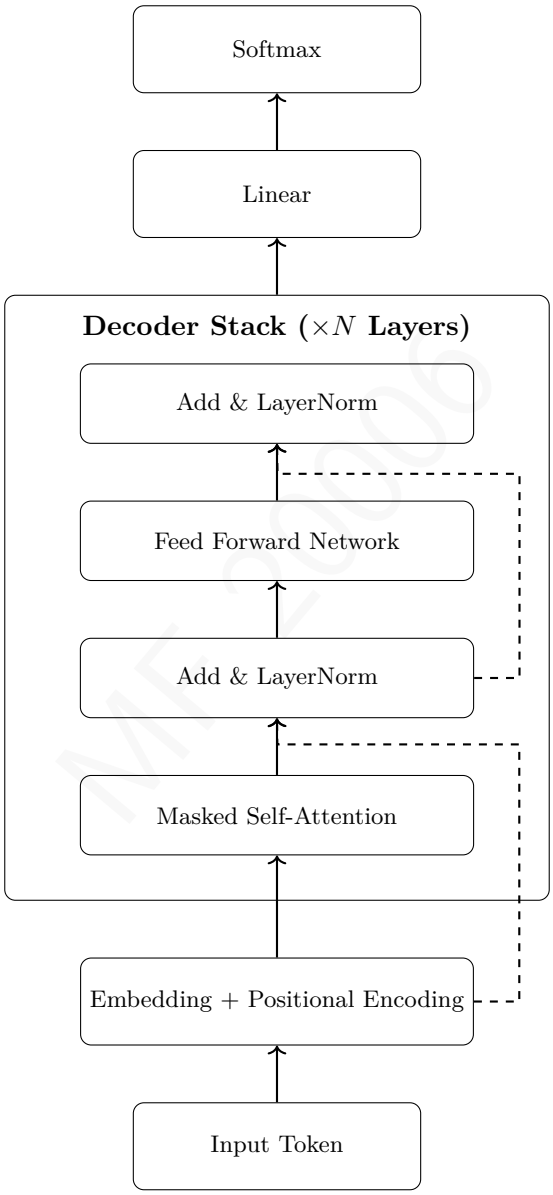


Figure 9.10: Architecture of a Decoder-only Transformer

As illustrated in Figure 9.10, a Decoder-only Transformer consists of multiple identical decoder blocks stacked together. Each block contains two main sublayers: *Masked Self-Attention* and a *Feed-Forward Network*. Residual connections and layer normalization are incorporated to facilitate the training of deep neural networks. Masked Self-Attention uses a causal mask to restrict information flow, ensuring that each token can attend only to itself and preceding tokens. This prevents access to future tokens and satisfies the requirements of autoregressive language modeling.