somewhat more challenging than the trivial 2D version used to produce the visualizations in this article. Below is a comparison of the performance for various algorithms discussed:

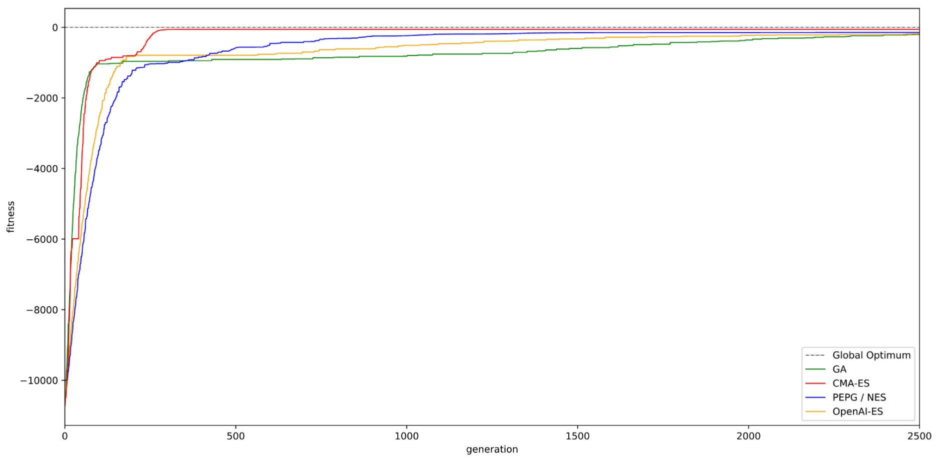

Figure 2.12: A comparison of the performance for various algorithms discussed in this Chapter for the 100-Dimensional Rastrigin function.

On this 100-D Rastrigin problem, none of the optimizers got to the global optimum solution, although CMA-ES comes close. CMA-ES is clearly the best performer. PGPE is in 2nd place, and OpenAI-ES / Genetic Algorithm falls behind. We had to use an annealing schedule to gradually lower σ for OpenAI-ES to make it perform better for this task.

```
[ 10.          10.          10.          10.          10.
  10.99495864  10.99495864   8.01008777  10.99495864   9.00504136
  10.          10.          10.           9.00504136  10.
   9.00504136  10.          10.99495864   8.01008776  10.99495864
  10.99495864  10.           9.00504137  10.99495863  10.99495864
   9.00504136  10.           9.00504136  10.99495863  10.
  10.99495863   8.01008776  10.99495863  10.          10.
  10.99495863   9.00504137  10.99495863   9.99999999  10.
  10.          10.          10.          10.          10.
  10.99495864  10.          10.           9.00504137   9.00504137
  10.           9.00504136  10.99495863  10.99495864  10.00000001
   9.00504136  10.           9.00504136  10.          10.
  10.           9.99999999  10.          10.           9.99999999
  10.           9.00504137  10.          10.           9.00504136
   9.00504137  10.           9.00504137   9.99999999  10.
   9.00504136   9.00504136  10.00000001  10.          10.
   9.00504136  10.99495864  10.          10.99495864  10.
  10.00000001  10.00000001  10.           9.00504136  10.99495863
   9.00504136  10.99495865  10.          10.          10.00000001
   9.99999999  10.          10.99495864   9.99999999  10.        ]
```

Figure 2.13: The final solution that CMA-ES discovered for 100-D Rastrigin function. The global optimal solution is a 100-dimensional vector of exactly 10.

2.2 Neural Networks

Artificial Neural Networks (ANNs) are a class of machine learning models loosely inspired by the structure and function of the human brain. They consist of layers of interconnected

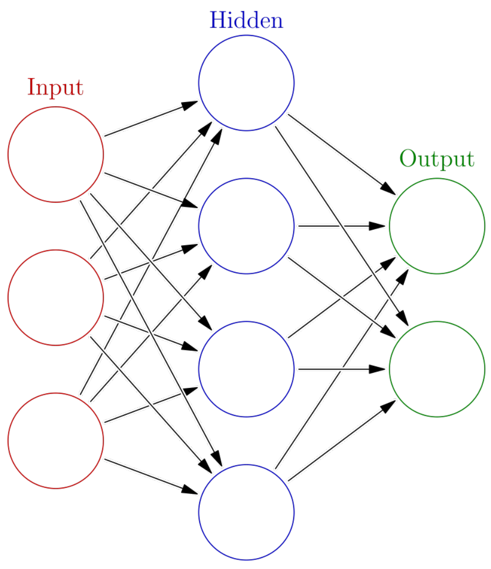

Figure 2.14: Feedforward Neural Network. This example nertwork has three inputs, one hidden layer with four nodes, and one output layer with two nodes. The input to the network propagates through the consecutive layers of the neural network to produce the outputs. Figure from wikipedia.

nodes or ”neurons” that process input data to produce an output. ANNs have shown remarkable success in various domains such as image recognition, natural language processing, and time-series forecasting. This chapter will provide the basic ideas behind the structure and function of neural networks, focusing on several key architectures used throughout the book: Feedforward Neural Networks (FNNs), Recurrent Neural Networks (RNNs), Long Short-Term Memory Networks (LSTMs), Convolutional Neural Networks (CNNs), and Transformers.

2.2.1 Feedforward Neural Networks (FNNs)

Feedforward Neural Networks are the simplest type of artificial neural network. They consist of an input layer, one or more hidden layers, and an output layer (Figure 2.14). Information flows in one direction, from the input to the output, without loops or cycles.

The network begins with the input layer, which receives raw data. Each node in this input layer corresponds to a feature or variable from the input dataset or the environment. This layer does not perform any calculations; it merely passes the input values to the next layer.

After the input layer, the data moves through one or more hidden layers. These layers are where the actual computations occur. Each hidden layer consists of multiple nodes, or neurons, which are fully connected to the nodes of the previous layer. Every connection between nodes has an associated weight that signifies the strength or importance of that connection. Each neuron also has a bias value that modifies the output.

For each neuron in a hidden layer, a weighted sum of all incoming inputs is calculated. This sum is then passed through an activation function , such as ReLU, Sigmoid, or Tanh, which introduces non-linearity to the model. The non-linearity is crucial because it allows the network to model more complex relationships between inputs and outputs. The output of the neurons in one layer becomes the input for the neurons in the next layer.

The final layer in the network is the output layer, which produces the network's prediction. The number of neurons in the output layer matches the number of possible outputs. For example, a binary classification task may have one or two output neurons, while a multi-class classification problem might have as many neurons as there are classes to predict.

An FNN can be represented mathematically as follows:

$$\mathbf{y} = \sigma(\mathbf{W}_h \cdot \sigma(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_h)$$

Here, $\mathbf{x}$ is the input vector, $\mathbf{W}_1$ and $\mathbf{W}_h$ are weight matrices for the first and hidden layers, respectively. The bias vectors are $\mathbf{b}_1$ and $\mathbf{b}_h$. The activation function, $\sigma(\cdot)$, is a non-linear function typically chosen to be a sigmoid, ReLU, or tanh. The output vector is denoted as $\mathbf{y}$.

2.2.2 Recurrent Neural Networks (RNNs)

A Recurrent Neural Network (RNN) (Figure 2.15,left) is a type of artificial neural network designed to recognize patterns in sequences of data, such as time series, text, or audio. Unlike feedforward neural networks, RNNs have connections that loop back, allowing information to persist. This architecture makes them particularly well-suited for tasks where context and order matter, enabling them to handle sequences of variable length and maintain a "memory" of what has been processed.

Let's have a look exactly how a recurrent neural network works. In the RNN, the neurons not only receive input from the previous layer but also from their previous states. This allows the network to maintain a form of memory about the past inputs, which is essential for tasks like speech recognition, machine translation, or any other problem where the current input is dependent on the previous inputs.

The network begins with an input layer that receives a sequence of data. Unlike feedforward networks, RNNs process sequences one element at a time. For example, in a text processing task, each word in a sentence might be fed into the network one by one.

The core of an RNN is its hidden layer, which is designed to maintain a hidden state, or memory, that captures information about the sequence. When an input element is fed into the network, it is combined with the previous hidden state to produce a new hidden state. Mathematically, this is often represented as:

$$h_t = f(W \cdot x_t + U \cdot h_{t-1} + b)$$

where:

- h_t represents the hidden state at time step t.
- x_t is the input at time step t.
- W and U are weight matrices for the input and hidden state, respectively.

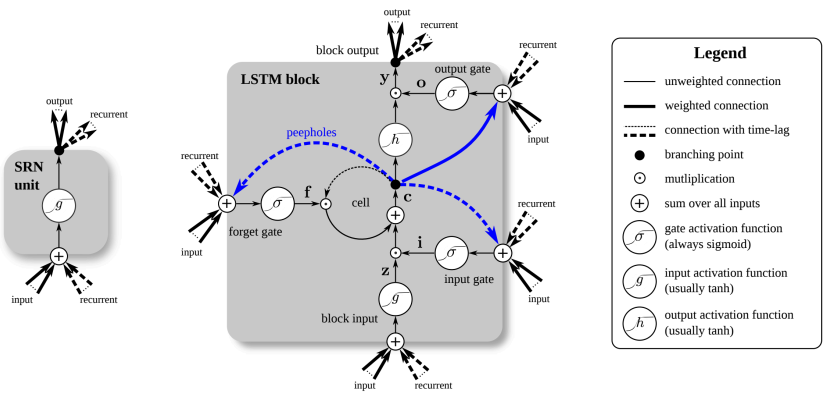

Figure 2.15: Left: Recurrent neural network. Right: Long Short-Term Memory. Figure from (Greff et al. 2016)

- b is a bias term.
- f is an activation function, typically a non-linear function like tanh or ReLU.

This hidden state is updated at each time step, capturing both the current input and the past context.

At each time step, the hidden state can produce an output, depending on the specific task. The output is computed using the current hidden state and a weight matrix. In a text prediction task, for example, the output at each time step might represent the predicted next word in a sentence.

In case of supervised learning problems, RNNs are typically trained using backpropagation through time (BPTT). However, they suffer from issues like vanishing and exploding gradients, which makes it difficult to capture long-term dependencies in the data.

2.2.3 Long Short-Term Memory Networks (LSTMs)

A Long Short-Term Memory (LSTM) network is a special type of Recurrent Neural Network (RNN) designed to overcome some of the limitations of traditional RNNs, particularly the problem of learning long-term dependencies (Figure 2.15, right). LSTMs are capable of learning and retaining information over extended periods, making them highly effective for tasks involving sequential data, such as language modeling, speech recognition, and time-series forecasting.

An LSTM network is composed of a series of LSTM cells, which replace the standard neurons in traditional RNNs. Each LSTM cell has a more complex internal structure designed to control the flow of information in and out of the cell, using several gates. These gates regulate which information is added, updated, or forgotten, allowing the network to maintain long-term dependencies and learn which pieces of information are important for making predictions.

An LSTM cell contains three main gates: the *forget gate*, the *input gate*, and the *output gate*. These gates use sigmoid activation functions to decide whether to let information pass through or not. Here's a breakdown of each component:

Forget Gate: The forget gate determines which parts of the cell's previous state should be discarded or "forgotten." It takes the current input (x_t) and the previous hidden state (h_{t-1}) and passes them through a sigmoid function. The output of this function is a value between 0 and 1 for each number in the cell state (C_{t-1}), where 0 represents "completely forget" and 1 represents "completely retain."

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Here:

- f_t is the forget gate's output.
- W_f is the weight matrix for the forget gate.
- b_f is the bias term for the forget gate.
- σ denotes the sigmoid function.

Input Gate: The input gate decides which new information will be added to the cell state. It consists of two parts: a sigmoid layer that determines which values will be updated and a tanh layer that creates a vector of new candidate values that could be added to the state. The results from these two layers are multiplied together to decide which new information to keep.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Here:

- i_t is the input gate's output.
- $\tilde{C}_t$ represents the new candidate values to be added.
- W_i and W_C are weight matrices for the input gate and candidate values.
- b_i and b_C are the bias terms for the input gate and candidate values.

Cell State Update: The new cell state C_t is updated by combining the old cell state C_{t-1} multiplied by the forget gate output f_t (which determines what to forget) and the new candidate values $\tilde{C}_t$ multiplied by the input gate output i_t (which determines what new information to add):

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

This equation effectively updates the cell state by retaining the necessary information from the past and incorporating the new relevant information.

Output Gate: The output gate determines the next hidden state h_t, which is used for the next time step and can also be an output for the current time step. The output gate first passes the current input and previous hidden state through a sigmoid function to decide

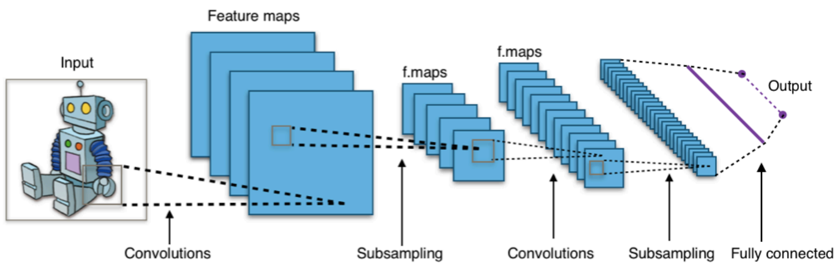

Figure 2.16: A typical architecture of a Convolutional Neural Network (CNN). The input image passes through multiple layers of convolutions, which extract various features, followed by subsampling (pooling) layers to reduce dimensionality. This process is repeated to create deeper feature maps, which are then flattened and connected to fully connected layers to generate the final output. Figure from Wikipedia.

which parts of the cell state to output. Then, it multiplies the cell state (after applying the tanh function to scale between -1 and 1) by the output of the sigmoid gate.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

Here:

- o_t is the output gate's output.
- h_t is the new hidden state.
- W_o is the weight matrix for the output gate.
- b_o is the bias term for the output gate.

The gating mechanisms in LSTM cells allow them to remember information for long periods. This is particularly useful in tasks where the context of earlier parts of a sequence is essential for making accurate predictions later. Additionally, LSTMs are specifically designed to mitigate the problem of vanishing gradients, which occurs when training traditional RNNs on long sequences. The cell state in LSTMs can maintain a more constant flow of gradients during backpropagation, allowing the network to learn long-term dependencies effectively.

2.2.4 Convolutional Neural Networks (CNNs)

A Convolutional Neural Network (CNN) is a type of deep learning model specifically designed to process and analyze data with a grid-like structure, such as images (Figure 2.16). CNNs are particularly effective for tasks that involve spatial hierarchies in data, such as image recognition, object detection, and video analysis. The architecture of CNNs is inspired by the visual cortex of the brain, where individual neurons respond to overlapping regions in the visual field.

A CNN consists of several layers, each with a specific function. The primary building blocks of a CNN are the *convolutional layers*, *pooling layers*, and *fully connected layers*. These layers work together to automatically and adaptively learn spatial hierarchies of features from input data.

Convolutional Layer: The convolutional layer is the core component of a CNN. It performs the convolution operation, which involves sliding a small filter or kernel (a matrix of weights) over the input data to produce a feature map. The filter captures spatial patterns such as edges, textures, or colors. This operation can be visualized as taking a small window of the input image, applying the filter, and generating an output value that represents a specific feature at that location.

Mathematically, the convolution operation can be expressed as:

$$(I * K)(x, y) = \sum_{i=1}^{m} \sum_{j=1}^{n} I(x+i, y+j) \cdot K(i, j)$$

where:

- I is the input image.
- K is the convolution kernel or filter.
- x, y are the coordinates of the pixel in the output feature map.
- m, n are the dimensions of the kernel.

The output of this operation is a set of feature maps that highlight specific patterns or features in the input data. Multiple filters can be used to detect different features, resulting in multiple feature maps.

Activation Function: After the convolutional layer, an activation function, typically the *Rectified Linear Unit (ReLU)*, is applied to introduce non-linearity. This non-linearity allows the network to learn complex patterns. The ReLU function is defined as:

$$f(x) = \max(0, x)$$

This activation function outputs the input directly if it is positive; otherwise, it outputs zero. It helps the network to learn non-linear relationships.

Pooling Layer: The pooling layer, also known as the subsampling or downsampling layer, reduces the spatial dimensions of the feature maps. This helps to reduce the number of parameters, computational complexity, and overfitting. The most common type of pooling is *max pooling*, which takes the maximum value from a small region of the feature map.

If the input to the pooling layer is a 2×2 window, max pooling selects the highest value from that window. Mathematically, max pooling over a region can be expressed as:

$$P(x, y) = \max\{f(i, j) : i, j \in \text{window}(x, y)\}$$

Here, $P(x, y)$ represents the output of the pooling operation at position (x, y), and $f(i, j)$ is the feature value at position (i, j).

Fully Connected Layer: After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers. In a fully connected layer, each neuron is connected to every neuron in the previous layer. The output of the final fully connected layer represents the class scores (in a classification problem) or other task-specific outputs.

The fully connected layer can be mathematically represented as:

$$y = W \cdot x + b$$

where:

- y is the output vector.
- W is the weight matrix.
- x is the input vector.
- b is the bias term.

In classification tasks, the output layer often uses a softmax activation function to convert the output scores into probabilities. The softmax function is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Here, z_i represents the output score for class i, and the denominator is the sum of the exponentials of all output scores. This function ensures that the output values are between 0 and 1 and sum to 1, representing a probability distribution over the classes.

2.2.5 Transformers

A Transformer (Vaswani et al. 2017) is a type of deep learning model that relies entirely on – what is called – a self-attention mechanisms to process input data, rather than traditional recurrent or convolutional layers. We will look at the self-attention mechanism in more detail below and then again in Chapter 4.4.1 in the context of indirect encodings. Transformers have become the foundation for many state-of-the-art models in natural language processing (NLP) and other fields, such as the GPT series, BERT, and more. They are particularly well-suited for handling sequential data and long-range dependencies, and they have demonstrated significant improvements in performance for tasks like machine translation, text generation, and summarization.

The Transformer architecture consists of an *encoder-decoder* structure, where both the encoder and decoder are composed of multiple layers of self-attention and feed-forward neural networks (Figure 2.17). The encoder takes an input sequence and processes it into an internal representation, which the decoder then uses to generate an output sequence. Each component in the Transformer leverages *self-attention* to weigh the importance of different elements in the input sequence and learn complex patterns.

Input Embedding and Positional Encoding: The input to a Transformer model is first converted into embeddings, which are fixed-length dense vector representations of the input tokens (words, subwords, etc.). Since Transformers do not inherently understand the order of the sequence, positional encodings are added to the embeddings to provide information about the relative positions of tokens in the sequence. The positional encodings use sine and cosine functions of different frequencies to create unique position vectors.

Mathematically, the positional encoding for a position *pos* and a dimension i is defined as:

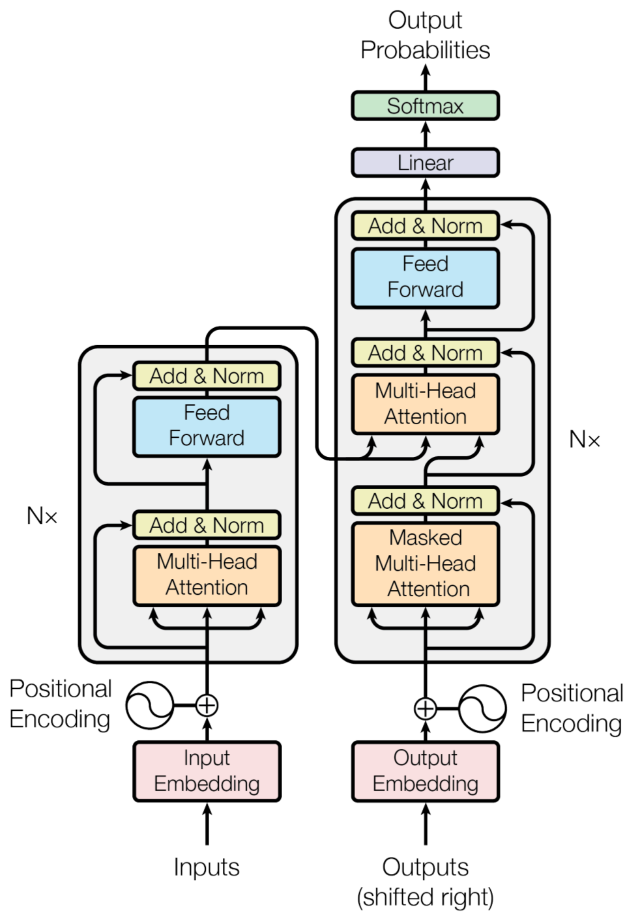

Figure 2.17: Illustration of the Transformer architecture, consisting of an encoder (left) and a decoder (right). The encoder is composed of a stack of layers, each containing a multi-head self-attention mechanism followed by a position-wise feed-forward network, with residual connections and layer normalization applied after each sub-layer. The decoder stack is similarly structured but includes an additional masked multi-head self-attention mechanism to prevent positions from attending to subsequent positions. Positional encodings are added to the input embeddings to provide information about the position of the words in the sequence. The final output is generated after applying a linear transformation and a softmax function to produce the output probabilities. Figure from Vaswani et al. 2017.

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \quad \mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

where:

- *pos* is the position of the token in the sequence.

- i is the dimension index.
- d_{model} is the dimension of the model's embedding space.

Self-Attention Mechanism: The core of the Transformer is the *self-attention* mechanism, which allows the model to focus on different parts of the input sequence when making predictions. Self-attention computes a weighted representation of each input token based on its relationship with all other tokens in the sequence. This is done by calculating three vectors: the *query (Q)*, *key (K)*, and *value (V)* vectors for each token. These vectors are derived using learned weight matrices:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

where:

- X is the input sequence.
- W^Q, W^K, W^V are weight matrices for the query, key, and value vectors, respectively.

The self-attention scores are computed by taking the dot product of the query and key vectors and scaling by the square root of the dimensionality of the key vectors. The scores are then passed through a softmax function to produce attention weights:

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

where:

- d_k is the dimension of the key vectors.

Multi-Head Attention: To allow the model to jointly attend to information from different representation subspaces, Transformers use *multi-head attention*. Instead of computing a single set of attention scores, the input is projected into multiple sets of queries, keys, and values, and the attention mechanism is applied in parallel. The outputs of these attention heads are concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Each head i performs the self-attention computation independently, and the results are combined to capture different aspects of the input data.

Feed-Forward Neural Network: After the multi-head attention layer, the output is passed through a position-wise *feed-forward neural network*. This consists of two linear transformations with a ReLU activation in between. The same feed-forward network is applied independently to each position in the sequence:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where:

- W_1, W_2 are weight matrices.
- b_1, b_2 are bias terms.

Layer Normalization and Residual Connections: To stabilize and speed up training, each sub-layer (multi-head attention and feed-forward neural network) is followed by a *layer normalization* step, which normalizes the output across the features. Additionally, the Transformer uses *residual connections* (skip connections) that add the input of each sub-layer to its output before applying layer normalization. This helps prevent the vanishing gradient problem and allows the model to learn more efficiently:

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

Stacking Layers: The encoder and decoder are composed of multiple identical layers (typically 6 to 12 in common implementations). Each encoder layer consists of a multi-head self-attention mechanism followed by a feed-forward neural network, while each decoder layer contains an additional cross-attention mechanism to attend to the encoder's output.

Output Decoding: The decoder generates the output sequence one token at a time. At each step, the decoder attends to all the previously generated tokens using masked self-attention (to prevent attending to future tokens) and to the encoder's output using a cross-attention mechanism. This process continues until the model generates a special end-of-sequence token.

2.3 Conclusion and End-Of-Chapter Questions

This chapter introduces the fundamental principles of evolutionary algorithms and neural networks, laying the foundation for their integration in neuroevolution. Evolutionary algorithms (EAs) are optimization techniques inspired by natural selection, operating on populations of candidate solutions that evolve over successive generations. Key processes include selection, mutation, and crossover, which allow populations to explore and exploit the search space for optimal or near-optimal solutions. The chapter discusses different types of EAs, such as genetic algorithms (GAs) and evolutionary strategies (ES), and their specific uses, advantages, and limitations in optimization problems.

Additionally, the chapter introduces neural networks, including basic architectures like feedforward and convolutional networks. These networks are designed to process and learn from data, enabling them to make decisions or predictions. When integrated with evolutionary algorithms, neural networks can evolve their structures and weights through mechanisms that bypass traditional gradient-based training. This integration can be particularly useful in tasks that are difficult to model with a clear objective function or gradient. By understanding the fundamentals of EAs and neural networks, readers will gain insight into how they can be combined to develop adaptive, resilient AI systems capable of tackling complex tasks.

Let's reinforce what you've learned in this chapter with a few questions designed to help you solidify key concepts and think critically about the basics of neuroevolution.

1. Define evolutionary algorithms and list their key components.
2. How do genetic algorithms work, and what role do crossover and mutation play?
3. Explain how fitness selection is performed in evolutionary algorithms.
4. Compare the roles of crossover and mutation in promoting diversity within a population.
5. What are some advantages and disadvantages of population-based optimization?

6. Describe how evolutionary algorithms handle multi-objective optimization problems.
7. Discuss the significance of mutation rate and crossover rate in evolutionary algorithms.
8. Explain how fitness landscapes affect the search process in evolutionary algorithms.
9. What challenges arise when applying evolutionary algorithms to neural networks, and how can these be addressed?

2.4 Chapter Review Questions

1. **Core Principles of Evolutionary Algorithms (EAs):** What are the key components of evolutionary algorithms? How do these components collectively emulate the process of natural selection?
2. **Genetic Algorithm (GA) Operations:** Describe the role of crossover and mutation in genetic algorithms, and explain how they contribute to maintaining diversity in the population.
3. **Challenges in Reinforcement Learning (RL):** Why are evolutionary strategies (ES) often used as an alternative to gradient-based optimization in reinforcement learning tasks, especially when dealing with sparse or noisy reward signals?
4. **Covariance Matrix Adaptation Evolution Strategy (CMA-ES):** How does CMA-ES adapt its search over successive generations? What advantage does this provide in comparison to simpler evolution strategies?
5. **Practical Applications of Fitness Shaping:** What is fitness shaping, and how does rank-based fitness shaping mitigate the impact of outliers in evolutionary optimization tasks?
6. **Feedforward Neural Networks (FNNs):** What is the primary purpose of the activation function in the hidden layers of a feedforward neural network? Why is non-linearity crucial for the network's performance?
7. **Recurrent Neural Networks (RNNs):** How do recurrent neural networks (RNNs) maintain information about past inputs? Why are they particularly well-suited for sequential data tasks like language modeling?
8. **Long Short-Term Memory Networks (LSTMs):** What are the roles of the forget, input, and output gates in an LSTM cell? How do they collectively help mitigate the vanishing gradient problem?
9. **Convolutional Neural Networks (CNNs):** Describe the purpose of the convolutional and pooling layers in a CNN. How do these layers work together to extract and summarize features from input data?
10. **Transformers:** What is the self-attention mechanism in a Transformer model? How does it enable the model to capture long-range dependencies in sequential data?