Training Report Day-22

1 July 2024

"ANN" typically stands for **Artificial Neural Network**. It's a type of computing system inspired by the biological neural networks of the human brain. ANNs are designed to recognize patterns, learn from data, and make predictions or classifications, making them widely used in machine learning and AI applications.

ANNs are composed of layers of nodes, or "neurons," that process input data and adjust their connections based on training data to minimize errors. The basic structure includes an input layer, one or more hidden layers, and an output layer. Variants include convolutional neural networks (CNNs) for image data and recurrent neural networks (RNNs) for sequential data.

1. Structure of an ANN

1. **Layers**: ANNs are structured in layers:
    - **Input Layer**: Receives the initial data, where each neuron represents a feature in the data.
    - **Hidden Layers**: These layers perform computations and transformations on the input data, extracting patterns and relationships.
    - **Output Layer**: Produces the final output, such as a classification label or prediction value.
2. **Neurons**: Each layer contains nodes (neurons) that hold weights, bias values, and an activation function. These components help the network learn and generalize from the input data.

2. Components of ANNs

- **Weights**: Parameters that adjust as the network learns, influencing how strongly input data affects the neuron’s output.
- **Biases**: Constants added to the weighted input to adjust the output further and help the model fit the data.

- **Activation Functions**: Functions applied to the neuron's output to introduce non-linearity, enabling the network to learn complex patterns. Common functions include ReLU (Rectified Linear Unit), Sigmoid, and Tanh.

3. Training Process

- **Forward Propagation**: The input data moves forward through the network to produce an output.
- **Loss Calculation**: The model's output is compared to the actual output, using a loss function to measure error.
- **Backpropagation**: The network adjusts weights and biases based on the error by moving backward through the network, using algorithms like gradient descent.

4. Types of ANNs

- **Feedforward Neural Networks**: The simplest form where data moves in one direction from input to output.
- **Convolutional Neural Networks (CNNs)**: Used primarily for image processing, CNNs apply filters to recognize spatial hierarchies in data.
- **Recurrent Neural Networks (RNNs)**: Suitable for sequence data, RNNs maintain memory of previous inputs to handle time-series data, language, or any data with temporal dependencies.
- **Generative Adversarial Networks (GANs)**: Consist of two networks, a generator and a discriminator, often used for creating realistic synthetic data.

5. Applications of ANNs

- **Image and Speech Recognition**: CNNs and RNNs are particularly effective for identifying patterns in visual and audio data.
- **Natural Language Processing (NLP)**: Tasks like sentiment analysis, machine translation, and text generation.
- **Medical Diagnosis**: ANNs can detect patterns in medical images and predict disease.
- **Financial Forecasting**: ANNs analyze historical data to predict stock prices, currency rates, or economic trends.
- **Self-Driving Cars**: They help in object detection, lane tracking, and decision-making.

6. Advantages of ANNs

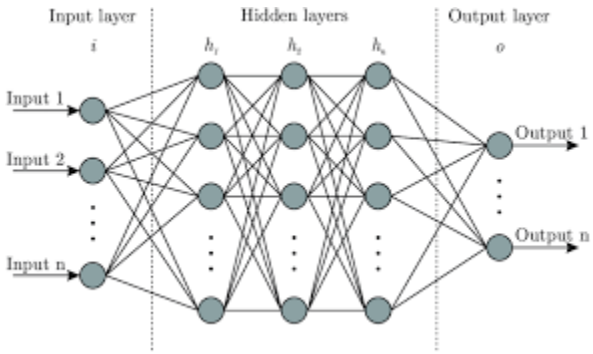
- **Flexibility**: Can adapt to various types of data, from structured to unstructured.
- **Accuracy**: Capable of achieving high accuracy in tasks with vast amounts of data.
- **Automatic Feature Extraction**: Unlike traditional algorithms, ANNs can automatically identify key features in data.

7. Limitations of ANNs

- **Data Requirement**: Require large amounts of data to train effectively.
- **Computational Power**: High training times and substantial computational resources are often needed.
- **Interpretability**: It can be challenging to understand how ANNs make decisions due to their complexity.

8. Training Algorithms

- **Gradient Descent**: The most common optimization algorithm used to minimize the error.
- **Stochastic Gradient Descent (SGD)**: A variant that updates weights more frequently for faster, albeit noisier, convergence.
- **Adam Optimizer**: Combines the advantages of both momentum and adaptive learning, widely used for better performance.




```
import numpy as np

# Define the activation function and its derivative
def sigmoid(x):
```

```
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Training data (XOR problem)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Seed random numbers for consistent results
np.random.seed(1)

# Initialize weights randomly
input_layer_neurons = 2  # Input layer (2 features)
hidden_layer_neurons = 2  # Hidden layer (2 neurons)
output_neurons = 1        # Output layer (1 neuron)

# Initialize weights and biases
weights_input_hidden = np.random.uniform(size=(input_layer_neurons, hidden_layer_neurons))
weights_hidden_output = np.random.uniform(size=(hidden_layer_neurons, output_neurons))
bias_hidden = np.random.uniform(size=(1, hidden_layer_neurons))
bias_output = np.random.uniform(size=(1, output_neurons))

# Training parameters
epochs = 10000
learning_rate = 0.1
# Training loop
for epoch in range(epochs):
    # Forward pass
    hidden_layer_input = np.dot(X, weights_input_hidden) + bias_hidden
    hidden_layer_activation = sigmoid(hidden_layer_input)
```

```
    output_layer_input = np.dot(hidden_layer_activation, weights_hidden_output) +
bias_output
    predicted_output = sigmoid(output_layer_input)

    # Calculate error
    error = y - predicted_output

    # Backpropagation
    d_predicted_output = error * sigmoid_derivative(predicted_output)
    error_hidden_layer = d_predicted_output.dot(weights_hidden_output.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_activation)

    # Update weights and biases
    weights_hidden_output += hidden_layer_activation.T.dot(d_predicted_output) *
learning_rate
    bias_output += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate
    weights_input_hidden += X.T.dot(d_hidden_layer) * learning_rate
    bias_hidden += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate

    # Print loss every 1000 epochs
    if epoch % 1000 == 0:
        print(f'Epoch {epoch}, Loss: {np.mean(np.abs(error))}')

# Output the final prediction
print("Final predicted output:", predicted_output)
```