INTEGRANTES:

GARCIA SANCHEZ SERGIO JESUS

VALENZUELA BERRELLEZA CESAR JESUS

NOMBRE DEL MAESTRO:

ZURIEL DATHAN MORA FELIX

MATERIA:

INTELIGENCIA ARTIFICIAL

FECHA DE ENTREGA:

09/03/2025

Paradigma Simbólico

Teoría y Características

Definición: El enfoque simbólico se basa en la manipulación de símbolos y reglas lógicas para representar el conocimiento y resolver problemas.

Fundamentos:

Utiliza lógica formal (por ejemplo, lógica proposicional o de primer orden).

El conocimiento se representa mediante reglas "SI-ENTONCES" (if-then).

Es explicable y transparente, ya que las decisiones se basan en reglas claras.

Aplicaciones comunes:

Sistemas expertos.

Procesamiento de lenguaje natural (NLP) basado en reglas.

Planificación y razonamiento automático.

Aplicaciones Prácticas

Ejemplo 1: Sistema Experto de Diagnóstico Médico

Descripción: Un sistema que diagnostica enfermedades basado en síntomas y reglas médicas.

Explicación: Las reglas simbólicas capturan el conocimiento de expertos médicos. Por ejemplo:

Regla: SI fiebre > 38°C Y dolor_garganta = verdadero ENTONCES diagnóstico = amigdalitis.

El sistema evalúa las condiciones de las reglas y aplica la inferencia lógica para llegar a un diagnóstico.

Beneficios:

Transparencia: Las decisiones son explicables y se pueden rastrear paso a paso.

Eficacia en dominios bien definidos: Funciona bien en áreas donde el conocimiento es claro y estructurado, como la medicina.

Bajo costo computacional: No requiere grandes recursos de hardware.

Limitaciones:

Incapacidad para manejar incertidumbre: No funciona bien con datos incompletos o ambiguos.

Rigidez: Las reglas deben ser actualizadas manualmente, lo que puede ser laborioso.

Escalabilidad limitada: No es adecuado para dominios con conocimiento complejo y dinámico.

Código de ejemplo:

```
class SistemaExperto:
    def __init__(self):
        self.reglas = [
            {"condiciones": ["fiebre", "dolor_garganta"], "diagnostico": "amigdalitis"},
            {"condiciones": ["fiebre", "tos"], "diagnostico": "gripe"}
        ]

    def diagnosticar (self, sintomas):
        for regla in self.reglas:
            if all(condicion in sintomas for condicion in regla["condiciones"]):
                return regla["diagnostico"]
        return "Diagnóstico no encontrado"

sistema = SistemaExperto()
sintomas = ["fiebre", "dolor_garganta"]
```

```
print("Diagnóstico:", sistema.diagnosticar(sintomas))
```

Ejemplo 2: Procesamiento de Lenguaje Natural (NLP) Basado en Reglas

Descripción: Un sistema que analiza texto utilizando reglas gramaticales.

Explicación: Las reglas simbólicas se utilizan para identificar estructuras gramaticales, como sujeto y predicado.

Por ejemplo, en la oración "El cielo es azul", la regla SI "es" está en la oración ENTONCES dividir en sujeto y predicado se aplica para extraer "El cielo" (sujeto) y "azul" (predicado).

Beneficios:

No requiere grandes cantidades de datos: Funciona bien con reglas definidas manualmente.

Útil para dominios específicos: Ideal para aplicaciones con lenguaje controlado, como chatbots simples.

Limitaciones:

No escala bien para lenguajes complejos: No maneja bien la ambigüedad o la variabilidad del lenguaje natural.

No aprende de nuevos datos: Las reglas deben ser actualizadas manualmente.

Código de ejemplo:

```
def analizar_oracion(oracion):
    if "es" in oracion:
        sujeto, predicado = oracion.split(" es ")
        return {"sujeto": sujeto, "predicado": predicado}
    return None

oracion = "El cielo es azul"
print("Análisis:", analizar_oracion(oracion))
```

Ejemplo 3: Sistema de Recomendación Basado en Reglas

Descripción: Un sistema que recomienda productos basado en preferencias del usuario.

Explicación: Las reglas simbólicas capturan las preferencias del usuario. Por ejemplo:

Regla: SI usuario compra "libros de ciencia ficción" ENTONCES recomendar "Dune".

El sistema aplica estas reglas para generar recomendaciones personalizadas.

Beneficios:

Personalización basada en reglas claras: Las recomendaciones son explicables y fáciles de entender.

Bajo costo computacional: No requiere entrenamiento de modelos complejos.

Limitaciones:

No aprende de nuevas preferencias: Las reglas deben ser actualizadas manualmente.

Escalabilidad limitada: No es adecuado para grandes catálogos de productos.

Código de ejemplo:

```
class Recomendador:
    def __init__(self):
        self.reglas = {
            "libros": ["Cien años de soledad", "1984"],
            "peliculas": ["El Padrino", "Interestelar"]
        }

    def recomendar (self, categoria):
        return self.reglas.get(categoria, ["No hay recomendaciones"])

recomendador = Recomendador()
print("Recomendaciones:", recomendador.recomendar("libros"))
```

Paradigma Conexionista

Teoría y Características

Definición: El enfoque conexionista se basa en redes neuronales artificiales que imitan el funcionamiento del cerebro humano. Especializado en tareas de aprendizaje automático.

Fundamentos:

Aprendizaje a partir de datos: Las redes neuronales aprenden patrones a partir de grandes cantidades de datos.

Capas de neuronas artificiales: Las redes están compuestas por capas de neuronas que procesan información de manera jerárquica.

Especialización: Ideal para tareas como reconocimiento de imágenes, procesamiento de lenguaje y predicción.

Aplicaciones comunes:

Reconocimiento facial.

Traducción automática.

Detección de fraudes.

Aplicaciones Prácticas

Ejemplo 1: Reconocimiento Facial con Redes Neuronales Convolucionales (CNN)

Descripción: Un sistema que identifica rostros en imágenes.

Explicación: Las CNN aprenden características jerárquicas de las imágenes, como bordes, texturas y formas.

Utilizan capas convolucionales para extraer características y capas densas para clasificar.

Beneficios:

Alta precisión en tareas de reconocimiento: Ideal para aplicaciones de seguridad y autenticación.

Aprendizaje automático de características: No requiere extracción manual de características.

Limitaciones:

Requiere grandes cantidades de datos etiquetados: El entrenamiento puede ser costoso.

Computacionalmente costoso: Requiere hardware especializado (GPUs).

Código de ejemplo:

```
from tensorflow.keras import layers, models


model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])


model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
print("Modelo de CNN creado.")
```

Ejemplo 2: Traducción Automática con Redes Neuronales Recurrentes (RNN)

Descripción: Un sistema que traduce texto entre idiomas.

Explicación: Las RNN procesan secuencias de palabras y aprenden correspondencias entre idiomas.

Utilizan capas recurrentes para capturar dependencias temporales en el texto.

Beneficios:

Traducción contextual y fluida: Ideal para aplicaciones como Google Translate.

Limitaciones:

Requiere grandes corpus de texto: El entrenamiento puede ser costoso.

Dificultad para traducir idiomas con pocos datos: No funciona bien con lenguajes minoritarios.

Código de ejemplo:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense


model = Sequential([
    Embedding(input_dim=10000, output_dim=64),
    LSTM(128),
    Dense(10000, activation='softmax')
])


model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy')
print("Modelo de traducción creado.")
```

Ejemplo 3: Detección de Spam con Redes Neuronales

Descripción: Un sistema que clasifica correos como spam o no spam.

Explicación: Aprende patrones de texto asociados con spam, como palabras clave o frases comunes.

Utiliza una red neuronal para clasificar los correos en dos categorías: spam o no spam.

Beneficios:

Alta precisión en clasificación: Ideal para filtrar correos no deseados.

Limitaciones:

Requiere datos etiquetados: El etiquetado manual puede ser laborioso.

Código de ejemplo:

```
from sklearn.neural_network import MLPClassifier
from sklearn.feature_extraction.text import CountVectorizer

correos = ["Gana dinero rápido", "Reunión mañana a las 10"]
etiquetas = [1, 0]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(correos)

model = MLPClassifier(hidden_layer_sizes=(10,), max_iter=1000)
model.fit(X, etiquetas)

print("Predicción:", model.predict(vectorizer.transform(["Oferta increíble"])))
```

Paradigma Computacional

Teoría y Características

Definición: El enfoque computacional se basa en algoritmos para resolver problemas específicos.

Fundamentos:

Técnicas de búsqueda: Algoritmos como A* para encontrar soluciones óptimas.

Optimización: Algoritmos genéticos para encontrar soluciones en espacios complejos.

Simulación: Métodos como Montecarlo para estimar resultados mediante muestreo aleatorio.

Aplicaciones comunes:

Planificación de rutas.

Optimización de recursos.

Simulación de sistemas complejos.

Aplicaciones Prácticas

Ejemplo 1: Algoritmo A* para Planificación de Rutas

Descripción: Encuentra la ruta más corta entre dos puntos.

Explicación: Utiliza una función heurística para guiar la búsqueda hacia el objetivo.

Combina el costo real del camino con una estimación heurística del costo restante.

Beneficios:

Eficiente en espacios de búsqueda grandes: Ideal para aplicaciones de navegación.

Limitaciones:

Depende de una heurística bien definida: Una heurística pobre puede afectar el rendimiento.

Código de ejemplo:

```
import heapq


def a_estrella(inicio, objetivo, heuristica):
    cola = [(0, inicio)]
    visitados = set()
    while cola:
        costo, actual = heapq.heappop(cola)
        if actual == objetivo:
            return costo
        visitados.add(actual)
        for vecino, costo_vecino in grafo[actual].items():
            if vecino not in visitados:
                heapq.heappush(cola, (costo + costo_vecino +
heuristica(vecino, objetivo), vecino))
    return float('inf')


grafo = {'A': {'B': 1, 'C': 4}, 'B': {'A': 1, 'C': 2}, 'C': {'A':
4, 'B': 2}}
print("Costo mínimo:", a_estrella('A', 'C', lambda x, y: 0))
```

Ejemplo 2: Algoritmo Genético para Optimización

Descripción: Simula la evolución para encontrar soluciones óptimas.

Explicación: Utiliza operadores como selección, cruce y mutación para evolucionar una población de soluciones.

Aplica la función de aptitud para evaluar y seleccionar las mejores soluciones.

Beneficios:

Útil para problemas complejos: Ideal para optimización en espacios no lineales.

Limitaciones:

Computacionalmente costoso: Requiere muchas iteraciones para converger.

Código de ejemplo:

```
import random


def funcion_objetivo(x):
    return x**2


poblacion = [random.uniform(-10, 10) for _ in range(10)]
for _ in range(100):
    poblacion = sorted(poblacion, key=funcion_objetivo)[:5]
    nuevos = [random.uniform(min(poblacion), max(poblacion)) for _
in range(5)]
    poblacion.extend(nuevos)
print("Mejor solución:", min(poblacion, key=funcion_objetivo))
```

Ejemplo 3: Simulación de Montecarlo

Descripción: Estima resultados mediante muestreo aleatorio.

Explicación: Utiliza repeticiones aleatorias para aproximar soluciones en problemas complejos.

Por ejemplo, estimar el valor de π mediante el lanzamiento aleatorio de puntos en un cuadrado.

Beneficios:

Flexible y aplicable a múltiples problemas: Ideal para problemas sin solución analítica.

Limitaciones:

Puede ser lento para alta precisión: Requiere muchas muestras para obtener resultados precisos.

Código de ejemplo:

```
import random


def montecarlo_pi(n):
    dentro = 0
    for _ in range(n):
        x, y = random.random(), random.random()
        if x**2 + y**2 <= 1:
            dentro += 1
    return 4 * dentro / n


print("Estimación de Pi:", montecarlo_pi(100000))
```

Referencias

Russell, S., & Norvig, P. (2021). Inteligencia Artificial: Un Enfoque Moderno. Pearson.

Chollet, F. (2018). Deep Learning with Python. Manning Publications.

Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly.

Documentación oficial de TensorFlow: https://www.tensorflow.org/.

Documentación oficial de Scikit-Learn: https://scikit-learn.org/.