ACERCA DEL AUTOR

Raúl Pedro Aceñero Eixarch

Ingeniero Informático y Licenciado en Derecho por la Universidad Jaime I de Castellón de la Plana. Profesor en el Ciclo Formativo de Grado Superior Desarrollo Aplicaciones Multiplataforma en la asignatura de "Programación Multimedia y Dispositivos Móviles", también imparte el Curso de Especialización Inteligencia Artificial y Big Data en la asignatura de "Programación de Inteligencia Artificial". Posee una experiencia docente de más de 25 años.

Además, ha investigado en Inteligencia Artificial para poder ayudar en el radiodiagnóstico de los nódulos cancerosos con redes neuronales e indicar en qué lugar de la radiografía pueda encontrarse el nódulo canceroso.

- *https://www.igi-global.com/gateway/article/283968*
- *https://www.coolmod.com/blog/secciones/ciencia/tecnologia-coolpc-a-disposicion-de-la-ciencia/*

Es autor del libro "Kotlin y Jetpack Compose. Desarrollo de aplicaciones Android", publicado por Editorial Ra-Ma.

INTRODUCCIÓN

Los smartphones son un hardware maravilloso que está extendido por todo el globo. Dada la cantidad que existen de todos ellos, cómo no aprovecharlos para extender la IA a través de ellos.

El libro quiere ser una herramienta que permita a los lectores llevar a cabo los proyectos que desean implementarlos con IA, al aglutinar las herramientas que nos ofrece la plataforma Android con su editor Android Studio con otras herramientas como OpenCV y Pytorch agrandan las posibilidades de los smartphones. No solamente, se ha quedado el libro en una conexión de internet para usar un LLM como Gemini con API, con LLaMA.cpp se consigue que el propio móvil en local pueda ejecutar un LLM que requeriría una GPU muy potente como puede ser Phi2 de Microsoft. Esto es debido al lenguaje C++ que, con su extraordinaria máxima eficiencia, bajo consumo de recursos y su gestión precisa permite aprovechar al máximo los recursos de hardware que se tienen.

Anteriormente el autor de este libro ha escrito: "Kotlin y Jetpack Compose. Desarrollo de aplicaciones Android". El cuál es el punto de partida para generar las interfaces que interactuaran con los elementos de la IA, haciendo que el usuario final logre comunicarse de una forma amigable con la IA. Para lograr este objetivo el libro se divide en varios capítulos como son:

- Capítulo 1 que está dedicado a conectarse con la API de Google para interactuar con Gemini y que tenga un frotntend amigable con el usuario de la aplicación.
- Capítulo 2 trata de usar con la cámara del móvil con OpenCV una biblioteca de código abierto especializada en visión por computadora y procesamiento de imágenes.

- Capítulo 3 se utilizará PyTorch es un framework de aprendizaje automático basada en la biblioteca Torch, utilizada para aplicaciones como visión artificial y procesamiento de lenguaje natural, desarrollada originalmente por Meta AI y ahora parte del paraguas de Linux Foundation. Es uno de los marcos de aprendizaje profundo más populares y que se utiliza mucho en investigación. Con lo cual se puede adaptar a las necesidades de cada proyecto.

- Capítulo 4 se estudiará YOLO (You Only Look Once) es un sistema de visión por ordenador para detectar objetos en tiempo real. YOLO puede localizar e identificar varios objetos en una imagen de una sola pasada. Para ello, puede usar tanto Pytorch como Tensorflow para la red neuronal convolucional.

- Capítulo 5 diseccionará ML Kit de Google proporciona las APIs de visión de aprendizaje automático integradas en el dispositivo para detectar rostros, escanear códigos de barras, etiquetar imágenes y mucho más. El Analizador de ML Kit facilita la integración del kit en la app de CameraX. En este caso se usará para detectar caracteres y detectar objetos.

- Capítulo 6 LLaMAC++ o LLaMA.cpp es una biblioteca de código abierto que permite ejecutar inferencias de modelos de lenguaje grande (LLMs) como Phi2, directamente en dispositivos locales utilizando C++, sin necesidad de dependencias externas. Su principal objetivo es ofrecer un rendimiento eficiente en una amplia variedad de hardware, incluidos dispositivos sin GPU dedicada como puede ser un móvil con la CPU ARM.

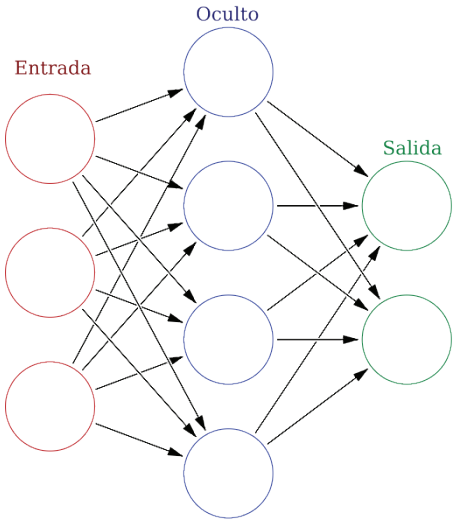
Para ello, el lector tendrá ejemplos prácticos por cada capítulo, estos son guiados paso a paso para que el lector los comprenda con detalle, fácil y completamente.

1

APP CON LLM GEMINI

1.1 INTRODUCCIÓN

En el Aprendizaje automático una red neuronal artificial[1] (abreviada ANN o NN) es un modelo dentro de los llamados sistemas conexionistas, inspirado en la estructura y función de las redes neuronales biológicas en los cerebros animales. Una ANN consta de unidades o nodos conectados llamados neuronas artificiales, que modelan vagamente las neuronas del cerebro. Estas están conectadas por bordes, que modelan las Sinapsis del cerebro. Cada neurona artificial recibe "señales" de las neuronas conectadas, luego las procesa y envía una señal a otras neuronas conectadas. La "señal" es un número real, y la salida de cada neurona se calcula mediante una función no lineal de la suma de sus entradas, llamada función de activación. La fuerza de la señal en cada conexión está determinada por un peso, que se ajusta durante el proceso de aprendizaje.



1_1

Por lo general, las neuronas se agrupan en capas. Las diferentes capas pueden realizar diferentes transformaciones en sus entradas. Las señales viajan desde la primera capa (la capa de entrada) hasta la última capa (la capa de salida), posiblemente pasando por múltiples capas intermedias (capas ocultas). Una red se denomina típicamente red neuronal profunda si tiene al menos dos capas ocultas.

Capacitación

Las redes neuronales se entrenan típicamente a través de la minimización de riesgos empíricos. Este método se basa en la idea de optimizar los parámetros de la red para minimizar la diferencia, o riesgo empírico, entre el resultado previsto y los valores objetivo-reales en un conjunto de datos determinado. Los métodos basados en gradientes, como la Retropropagación o Propagación hacía atrás, se utilizan generalmente para estimar los parámetros de la red. Durante la fase de entrenamiento, las ANN aprenden de los datos de entrenamiento etiquetados, actualizando iterativamente sus parámetros para minimizar una Función de pérdida definida. Las redes neuronales artificiales se utilizan para diversas tareas, como el modelado predictivo, el control adaptativo y la resolución de problemas en el ámbito de la inteligencia artificial. Pueden aprender de la experiencia y extraer conclusiones de un conjunto de información complejo y aparentemente no relacionado. Sobresalen en áreas donde la detección de soluciones o características es difícil de expresar con la programación convencional. Para realizar este aprendizaje automático, normalmente, se intenta minimizar una función de pérdida que evalúa la red en su total. Los valores de los pesos de las neuronas

se van actualizando, buscando reducir el valor de la función de pérdida. Este proceso se realiza mediante la propagación hacia atrás.

Red neuronal transformadora o Transformer

Un **Transformader**[2][3] es una arquitectura de aprendizaje profundo desarrollada por investigadores de Google y basada en el mecanismo de atención de múltiples cabezas, propuesto en un artículo de 2017 " Attention Is All You Need ". El texto se convierte en representaciones numéricas llamadas tokens, y cada token se convierte en un vector al buscar en una tabla de incrustación de palabras. En cada capa, cada token se contextualiza dentro del alcance de la ventana de contexto con otros tokens (sin máscara) a través de un mecanismo de atención de múltiples cabezas paralelo que permite que la señal de los tokens clave se amplifique y los tokens menos importantes se reduzcan.

Tokenización[4]

Como los algoritmos de aprendizaje automático procesan números en lugar de texto, el texto debe convertirse en números. En el primer paso, se decide un vocabulario, luego se asignan índices enteros de forma arbitraria pero única a cada entrada de vocabulario y, finalmente, se asocia una incrustación al índice entero. Los algoritmos incluyen codificación de pares de bytes (BPE) y WordPiece. También hay tokens especiales que sirven como caracteres de control, como [MASK] para el token enmascarado (como se usa en BERT) y [UNK] ("desconocido") para caracteres que no aparecen en el vocabulario. Además, se usan algunos símbolos especiales para indicar un formato de texto especial. Por ejemplo, "Ġ" indica un espacio en blanco anterior en RoBERTa y GPT. "##" indica la continuación de una palabra anterior en BERT.

Por ejemplo, el tokenizador BPE utilizado por GPT-3 (Legacy) se dividiría tokenizer: texts -> series of numerical "tokens" como:

token	izer	:	texts	->	series	of	numerical	"	t	ok	ens	"

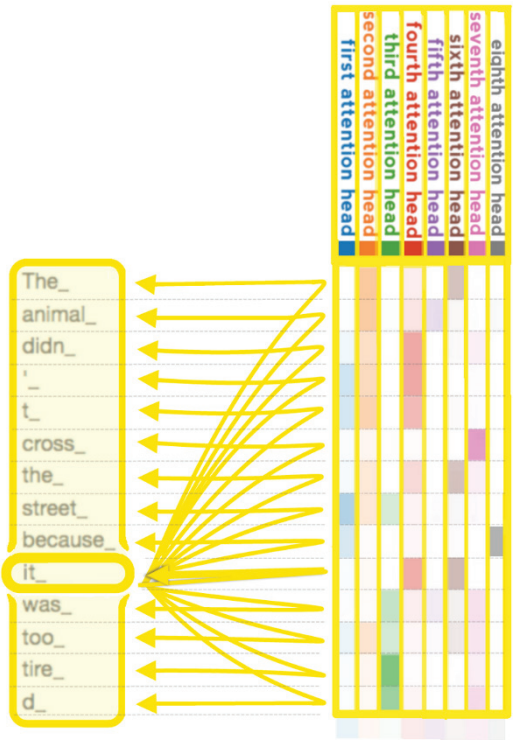
La tokenización también comprime los conjuntos de datos. Debido a que los LLM generalmente requieren que la entrada sea una matriz que no esté escalonada, los textos más cortos deben "rellenarse" hasta que coincidan con la longitud del más largo. La cantidad de tokens que se necesitan, en promedio, por palabra depende del idioma del conjunto de datos.

Para averiguar qué tokens son relevantes entre sí dentro del alcance de la ventana de contexto, el mecanismo de atención calcula pesos "suaves" para cada token, más precisamente para su incrustación, utilizando múltiples cabezas de atención, cada una con su propia "relevancia" para calcular sus propios pesos suaves. Por ejemplo,

el modelo GPT-2 pequeño (es decir, con un tamaño de parámetro de 117 millones) tenía doce cabezas de atención y una ventana de contexto de solo 1k tokens. En su versión mediana tiene 345 millones de parámetros y contiene 24 capas, cada una con 12 cabezas de atención. Para el entrenamiento con descenso de gradiente se utilizó un tamaño de lote de 512.

Los modelos más grandes, como Gemini 1.5 de Google, presentado en febrero de 2024, pueden tener una ventana de contexto de hasta 1 millón (la ventana de contexto de 10 millones también fue "probada con éxito"). Otros modelos con ventanas de contexto grandes incluyen Claude 2.1 de Anthropic, con una ventana de contexto de hasta 200k tokens. Tenga en cuenta que este máximo se refiere al número de tokens de entrada y que el número máximo de tokens de salida difiere de la entrada y, a menudo, es menor. Por ejemplo, el modelo GPT-4 Turbo tiene una salida máxima de 4096 tokens.

Cuando cada cabeza calcula, según su propio criterio, cuánto otros tokens son relevantes para el token "it_", note que la segunda cabeza de atención, representada por la segunda columna, se enfoca más en las primeras dos filas, es decir, los tokens "The" y "animal", mientras que la tercera columna se enfoca más en las dos filas inferiores, es decir, en "tired", que se ha tokenizado en dos tokens.



1_3

La longitud de una conversación que el modelo puede tener en cuenta al generar su próxima respuesta también está limitada por el tamaño de una ventana de contexto. Si la longitud de una conversación, por ejemplo con ChatGPT, es mayor que su ventana de contexto, solo se tienen en cuenta las partes dentro de la ventana de contexto al generar la próxima respuesta, o el modelo debe aplicar algún algoritmo para resumir las partes demasiado distantes de la conversación.

Las desventajas de hacer una ventana de contexto más grande incluyen un mayor costo computacional y posiblemente diluir el enfoque en el contexto local, mientras que hacerla más pequeña puede hacer que un modelo pase por alto una importante dependencia de largo alcance. Equilibrarlas es una cuestión de experimentación y consideraciones específicas del dominio.

Un modelo puede ser entrenado previamente para predecir cómo continúa el segmento o qué falta en el segmento, dado un segmento de su conjunto de datos de entrenamiento. Puede ser autorregresivo (es decir, predecir cómo continúa el segmento, como lo hacen los GPT): por ejemplo, dado un segmento "I like to eat", el modelo predice " ice cream" o "sushi".

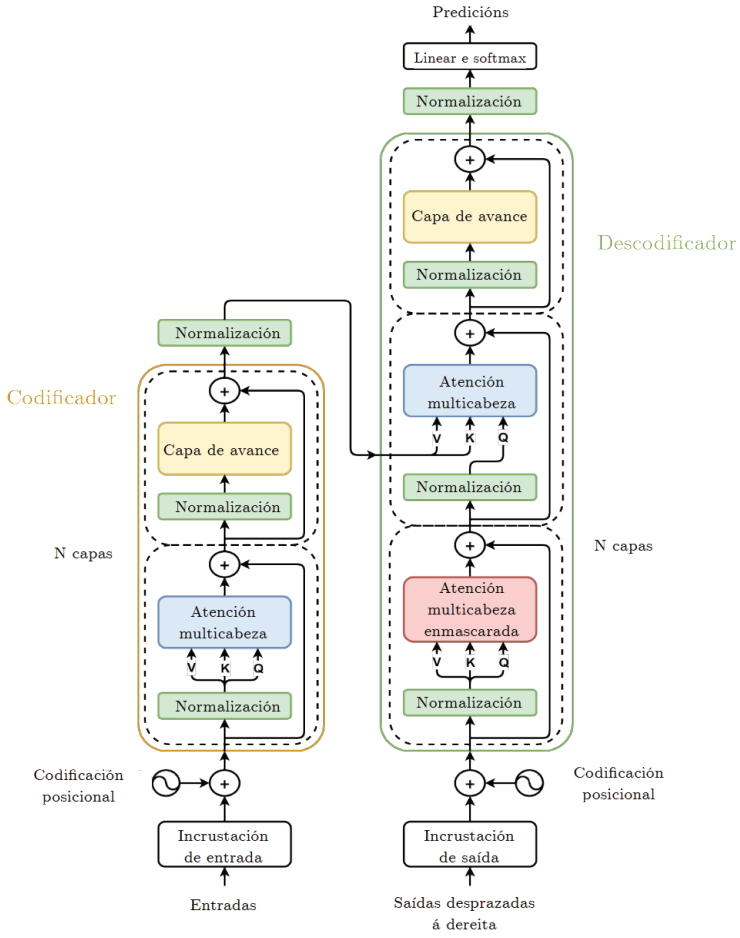
"enmascarado" (es decir, rellenando las partes faltantes del segmento, como lo hace "BERT"): por ejemplo, dado un segmento "I like to [__] [__] cream", el modelo predice que faltan "eat" and "ice".

Los modelos pueden entrenarse en tareas auxiliares que prueban su comprensión de la distribución de datos, como la predicción de la siguiente oración (NSP), en la que se presentan pares de oraciones y el modelo debe predecir si aparecen consecutivamente en el corpus de entrenamiento. Durante el entrenamiento, la pérdida de regularización también se utiliza para estabilizar el entrenamiento. Sin embargo, la pérdida de regularización generalmente no se utiliza durante las pruebas y la evaluación.

Por lo tanto, los transformadores tienen la ventaja de no tener unidades recurrentes y, por lo tanto, requieren menos tiempo de entrenamiento que las arquitecturas neuronales recurrentes (RNN) anteriores, como la memoria a corto plazo larga (LSTM). Las variaciones posteriores se han adoptado ampliamente para entrenar modelos de lenguaje grandes (LLM) en grandes conjuntos de datos (de lenguaje), como el corpus de Wikipedia y Common Crawl.

Los transformadores se desarrollaron por primera vez como una mejora de las arquitecturas anteriores para la traducción automática, pero han encontrado muchas aplicaciones desde entonces. Se utilizan en el procesamiento del lenguaje natural a gran escala, la visión por computadora (transformadores de visión), el aprendizaje de refuerzo, el audio, el procesamiento multimodal, la robótica, e incluso el

ajedrez. También ha llevado al desarrollo de sistemas preentrenados, como los transformadores preentrenados generativos (GPT) y BERT (representaciones de codificador bidireccional a partir de transformadores).



1_2

Aplicaciones de LLM

El transformador ha tenido un gran éxito en el procesamiento del lenguaje natural (PLN). Muchos modelos de large language models o LLM , como GPT-2, GPT-3, GPT-4, AlbertAGPT, Claude, BERT, XLNet, RoBERTa y ChatGPT, demuestran la capacidad de los transformadores para realizar una amplia variedad de subtareas

relacionadas con el PLN y sus aplicaciones prácticas o del mundo real relacionadas, entre ellas:

- Traducción automática.
- Predicción de series temporales.
- Resumen de documentos.
- Generación de documentos.
- Reconocimiento de entidades nombradas (NER).
- Escribir código de computadora basado en requisitos expresados en lenguaje natural.
- Conversión de voz a texto.

Más allá del PNL tradicional, la arquitectura del transformador ha tenido éxito en otras aplicaciones, como:

- Análisis de secuencia biológica.
- Comprensión del vídeo.
- Plegamiento de proteínas (como AlphaFold).
- Evaluación de posiciones del tablero de ajedrez. Utilizando únicamente la evaluación estática (es decir, sin búsqueda Minimax), un Transformer logró un sistema de clasificación Elo de 2895, lo que lo coloca en el nivel de un gran maestro.

1.2 GEMINI

Gemini[5] (antes conocida como Bard) es una familia de modelos de lenguaje multimodales de gran tamaño desarrollados por Google DeepMind, que sirven como sucesores de LaMDA y PaLM 2. Compuesto por Gemini Ultra, Gemini Pro, Gemini Flash y Gemini Nano, fue anunciado el 6 de diciembre de 2023, posicionado como un competidor de GPT-4 de OpenAI.

El aprendizaje multimodal[6] es un tipo de aprendizaje profundo que integra y procesa múltiples tipos de datos, denominados modalidades , como texto, audio, imágenes o vídeo. Esta integración permite una comprensión más holística de datos complejos, lo que mejora el rendimiento del modelo en tareas como la respuesta visual a preguntas, la recuperación intermodal, la generación de texto a imagen, la clasificación estética, y el subtitulado de imágenes.

Los modelos multimodales de gran tamaño, como Google Gemini y GPT-4o, se han vuelto cada vez más populares desde 2023, lo que permite una mayor versatilidad y una comprensión más amplia de los fenómenos del mundo real.

Los transformadores también se pueden usar/adaptar para modalidades (entrada o salida) más allá del texto, generalmente encontrando una forma de "tokenizar" la modalidad.

Los modelos multimodales se pueden entrenar desde cero o mediante un ajuste fino. Un estudio de 2022 descubrió que los Transformers preentrenados solo en lenguaje natural se pueden ajustar en solo el 0,03 % de los parámetros y volverse competitivos con los LSTM en una variedad de tareas lógicas y visuales, lo que demuestra el aprendizaje por transferencia. LLaVA era un modelo de visión-lenguaje compuesto por un modelo de lenguaje (Vicuna-13B) y un modelo de visión (ViT -L/14), conectados por una capa lineal. Solo la capa lineal está ajustada.

Los transformadores de visión adaptan el transformador a la visión por computadora descomponiendo las imágenes de entrada como una serie de parches, convirtiéndolos en vectores y tratándolos como tokens en un transformador estándar.

Conformer y posteriormente Whisper siguen el mismo patrón para el reconocimiento de voz, primero convirtiendo la señal de voz en un espectrograma, que luego se trata como una imagen, es decir, se descompone en una serie de parches, se convierte en vectores y se trata como fichas en un transformador estándar.

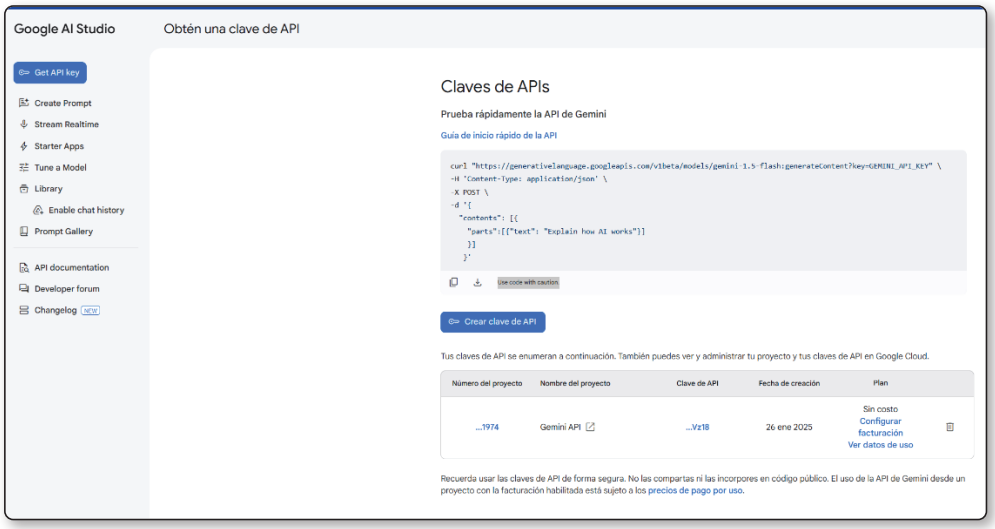
Los perceptores son una variante de los transformadores diseñados para la multimodalidad.

Para la generación de imágenes, las arquitecturas notables son DALL-E 1 (2021), Parti (2022), Phenaki (2023), y Muse (2023). A diferencia de los modelos posteriores, DALL-E no es un modelo de difusión. En cambio, utiliza un Transformer de solo decodificador que genera autorregresivamente un texto, seguido de la representación de token de una imagen, que luego es convertida por un autocodificador variacional en una imagen. Parti es un Transformer de codificador-decodificador, donde el codificador procesa un mensaje de texto y el decodificador genera una representación de token de una imagen. Muse es un Transformer de solo codificador que está entrenado para predecir tokens de imagen enmascarados a partir de tokens de imagen desenmascarados. Durante la generación, todos los tokens de entrada están enmascarados y las predicciones de mayor confianza se incluyen para la siguiente iteración, hasta que se predicen todos los tokens. Phenaki es un modelo de texto a vídeo. Es un transformador enmascarado bidireccional condicionado a tokens de texto precalculados. Los tokens generados se decodifican luego en un vídeo.

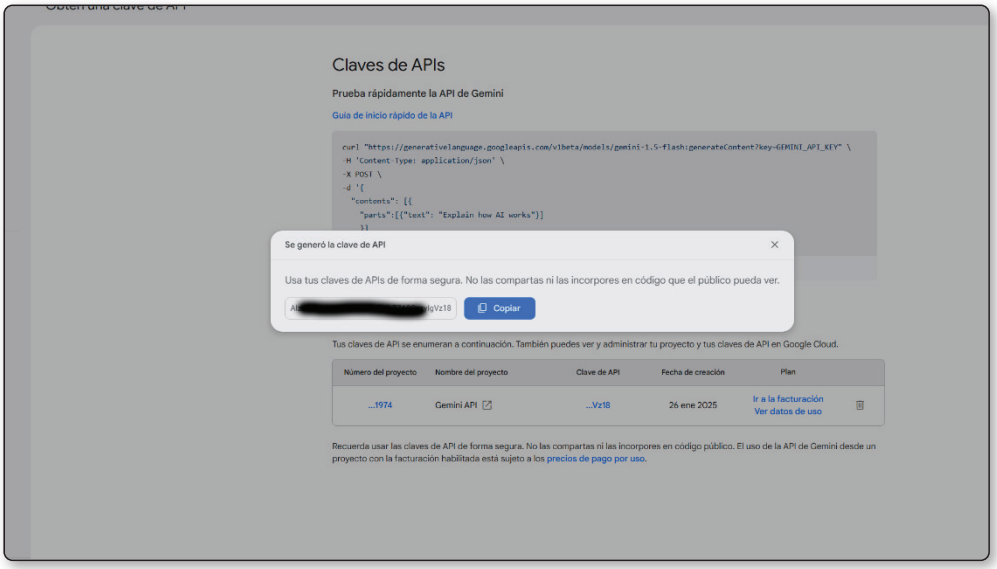
1.3 INSTALACIÓN DE GEMINI EN UN MÓVIL ANDROID

Para poder usar Gemini desde una aplicación propia se debe acceder a:

https://aistudio.google.com/app/apikey



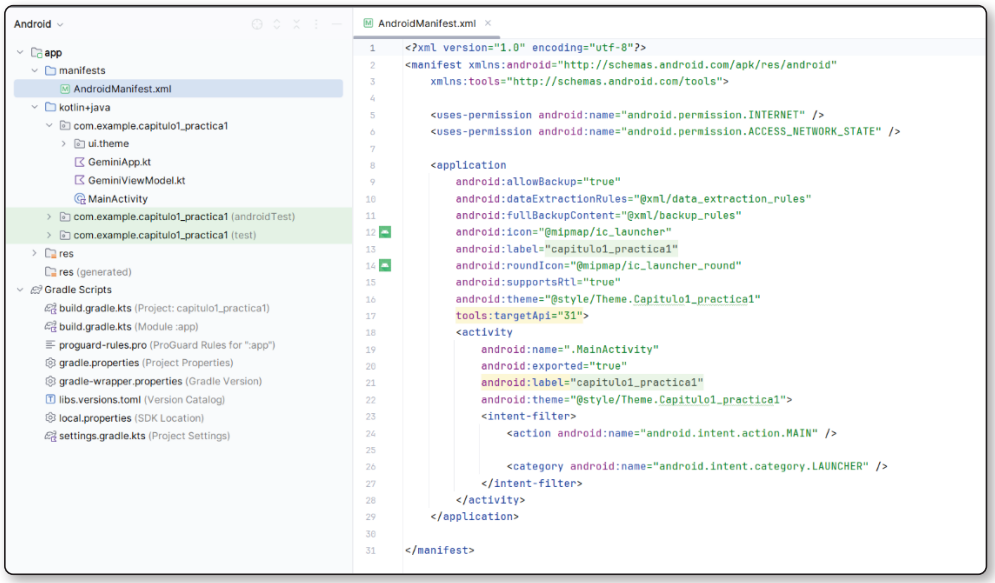
Pulsar Crear calve de API



Crear la aplicación: integración de la API de Gemini en una aplicación de dialogo con un LLM.

En este caso se llamará “capitulo1_practica1”

Paso 1: añadir permisos en AndroidManifest.xml



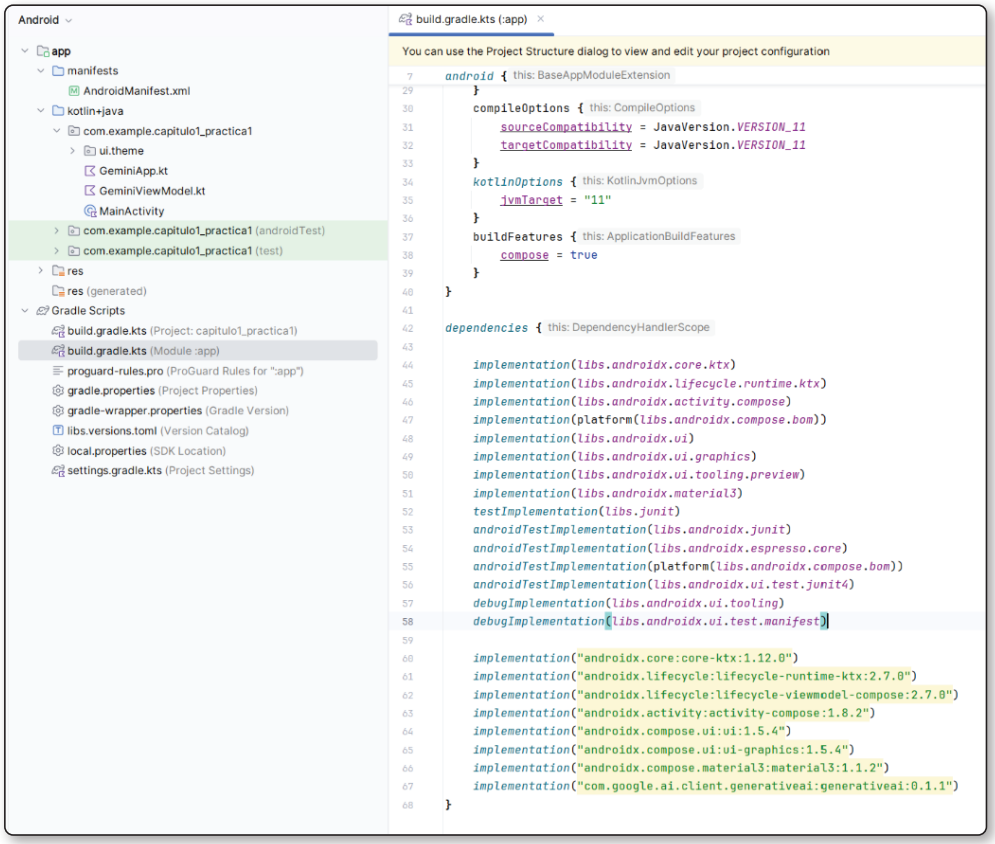
1_6

AndroidManifest.xml:

```
<?xml version=”1.0” encoding=”utf-8”?>
<manifest xmlns:android=”http://schemas.android.com/apk/res/android”
    xmlns:tools=”http://schemas.android.com/tools”>

    <uses-permission android:name=”android.permission.INTERNET” />
    <uses-permission android:name=”android.permission.ACCESS_NETWORK_STATE” />

    <application
        android:allowBackup=”true”
        android:dataExtractionRules=”@xml/data_extraction_rules”
        android:fullBackupContent=”@xml/backup_rules”
        android:icon=”@mipmap/ic_launcher”
        android:label=”@string/app_name”
        android:roundIcon=”@mipmap/ic_launcher_round”
        android:supportsRtl=”true”
        android:theme=”@style/Theme.Capitulo1_practica1”
```

```
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.Capitulo1_practica1">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Paso 2: añadir las dependencias en **build.gradle.kts** (Module:app)



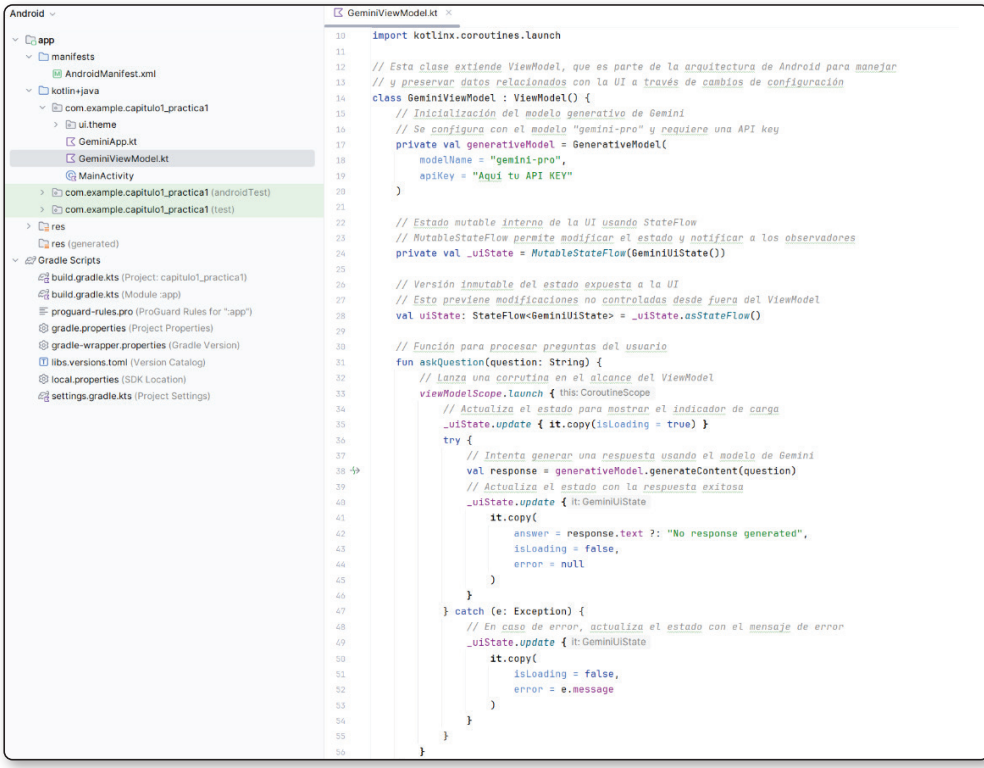
1_7

```
implementation("androidx.core:core-ktx:1.12.0")
implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.7.0")
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")
implementation("androidx.activity:activity-compose:1.8.2")
implementation("androidx.compose.ui:ui:1.5.4")
implementation("androidx.compose.ui:ui-graphics:1.5.4")
implementation("androidx.compose.material3:material3:1.1.2")
implementation("com.google.ai.client.generativeai:generativeai:0.1.1")
```

Paso 3: crear la clase **GeminiViewModel**

Este código implementa el patrón MVVM (Model-View-ViewModel) para manejar la interacción con la API de Gemini. El ViewModel actúa como una capa intermedia entre la interfaz de usuario y la lógica de negocio, gestionando el estado de la aplicación de manera segura y eficiente. Utiliza Kotlin Coroutines y Flow para manejar operaciones asíncronas y actualizaciones de estado de manera reactiva.

La clase GeminiUiState encapsula todo el estado necesario para la interfaz de usuario, permitiendo actualizaciones atómicas y manteniendo la consistencia de los datos. Esto facilita la gestión del estado de la aplicación y permite una clara separación de responsabilidades entre la UI y la lógica de negocio.



1_8

GeminiViewModel.kt:

```
package com.example.capitulo1_practica1

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.google.ai.client.generativeai.GenerativeModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch

// Esta clase extiende ViewModel, que es parte de la arquitectura de Android
para manejar
// y preservar datos relacionados con la UI a través de cambios de configuración
class GeminiViewModel : ViewModel() {
    // Inicialización del modelo generativo de Gemini
    // Se configura con el modelo "gemini-pro" y requiere una API key
    private val generativeModel = GenerativeModel(
        modelName = "gemini-pro",
        apiKey = "Aquí tu API KEY"
    )

    // Estado mutable interno de la UI usando StateFlow
    // MutableStateFlow permite modificar el estado y notificar a los observadores
    private val _uiState = MutableStateFlow(GeminiUiState())

    // Versión inmutable del estado expuesta a la UI
    // Esto previene modificaciones no controladas desde fuera del ViewModel
    val uiState: StateFlow<GeminiUiState> = _uiState.asStateFlow()

    // Función para procesar preguntas del usuario
    fun askQuestion(question: String) {
        // Lanza una corrutina en el alcance del ViewModel
        viewModelScope.launch {
            // Actualiza el estado para mostrar el indicador de carga
            _uiState.update { it.copy(isLoading = true) }
            try {
                // Intenta generar una respuesta usando el modelo de Gemini
                val response = generativeModel.generateContent(question)
                // Actualiza el estado con la respuesta exitosa
                _uiState.update {
                    it.copy(
                        answer = response.text ?: "No response generated",
                        isLoading = false,
                        error = null
                    )
                }
            } catch (e: Exception) {
```

```
                // En caso de error, actualiza el estado con el mensaje de error
                _uiState.update {
                    it.copy(
                        isLoading = false,
                        error = e.message
                    )
                }
            }
        }
    }
}

// Clase de datos que representa el estado de la UI
// Contiene toda la información necesaria para mostrar la interfaz de usuario
data class GeminiUiState(
    val question: String = "",  // Pregunta actual
    val answer: String = "",    // Respuesta del modelo
    val isLoading: Boolean = false, // Indicador de carga
    val error: String? = null   // Mensaje de error si algo falla
)
```

Paso 4: crear la clase

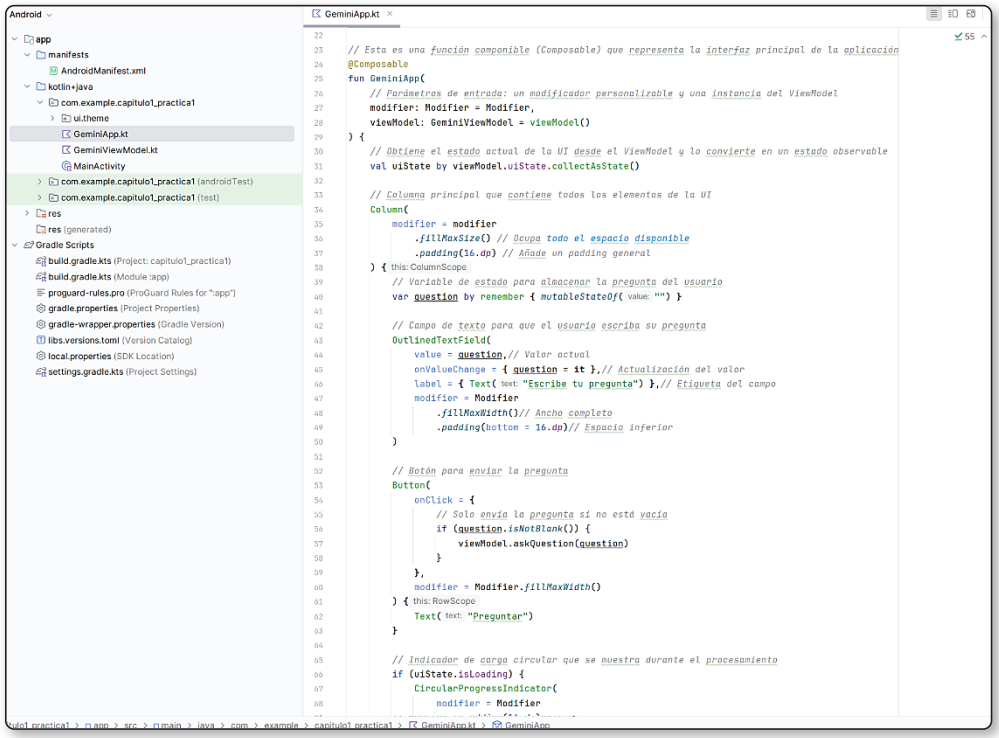
Esta interfaz de usuario está construida utilizando Jetpack Compose, el moderno toolkit de UI declarativa de Android. La función GeminiApp actúa como el contenedor principal de la aplicación y organiza los diferentes elementos de la interfaz de manera vertical utilizando una Column.

La interfaz implementa un patrón de diseño reactivo donde los cambios en el estado (gestionado por el ViewModel) se reflejan automáticamente en la UI. Incluye manejo de estados de carga, errores y la visualización de respuestas, proporcionando una experiencia de usuario completa y fluida.

Los elementos principales de la interfaz son:

- Un campo de texto para introducir preguntas.
- Un botón para enviar la pregunta.
- Un indicador de carga durante el procesamiento.
- Visualización de errores si ocurren.
- Visualización de la respuesta del modelo Gemini.

Cada elemento está cuidadosamente espaciado y estilizado usando modificadores de Compose, siguiendo las guías de diseño de Material Design 3.



1_9

GeminiApp.kt:

```
package com.example.capitulo1_practica1

import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.CircularProgressIndicator
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
```

```
import androidx.compose.ui.unit.dp
import androidx.lifecycle.viewmodel.compose.viewModel

// Esta es una función componible (Composable) que representa la interfaz
principal de la aplicación
@Composable
fun GeminiApp(
    // Parámetros de entrada: un modificador personalizable y una instancia del
ViewModel
    modifier: Modifier = Modifier,
    viewModel: GeminiViewModel = viewModel()
) {
    // Obtiene el estado actual de la UI desde el ViewModel y lo convierte en un
estado observable
    val uiState by viewModel.uiState.collectAsState()

    // Columna principal que contiene todos los elementos de la UI
    Column(
        modifier = modifier
            .fillMaxSize() // Ocupa todo el espacio disponible
            .padding(16.dp) // Añade un padding general
    ) {
        // Variable de estado para almacenar la pregunta del usuario
        var question by remember { mutableStateOf("") }

        // Campo de texto para que el usuario escriba su pregunta
        OutlinedTextField(
            value = question,// Valor actual
            onValueChange = { question = it },// Actualización del valor
            label = { Text("Escribe tu pregunta") },// Etiqueta del campo
            modifier = Modifier
                .fillMaxWidth()// Ancho completo
                .padding(bottom = 16.dp)// Espacio inferior
        )

        // Botón para enviar la pregunta
        Button(
            onClick = {
                // Solo envía la pregunta si no está vacía
                if (question.isNotBlank()) {
                    viewModel.askQuestion(question)
                }
            },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text("Preguntar")
        }

        // Indicador de carga circular que se muestra durante el procesamiento
        if (uiState.isLoading) {
            CircularProgressIndicator(
                modifier = Modifier
                    .padding(16.dp)
                    .align(Alignment.CenterHorizontally)
```

```
            )
        }

        // Muestra mensaje de error si existe
        if (uiState.error != null) {
            Text(
                text = “Error: ${uiState.error}”,
                color = MaterialTheme.colorScheme.error,// Color rojo para
errores
                modifier = Modifier.padding(top = 16.dp)
            )
        }

        // Muestra la respuesta si existe
        if (uiState.answer.isNotBlank()) {
            Text(
                text = uiState.answer,
                modifier = Modifier.padding(top = 16.dp)
            )
        }
    }
}
```

Paso 5: llamar a la función desde MainActivity.kt

Esta clase MainActivity es el punto de entrada principal de la aplicación Android. Implementa la configuración básica necesaria para una aplicación Jetpack Compose y establece la estructura fundamental de la interfaz de usuario.

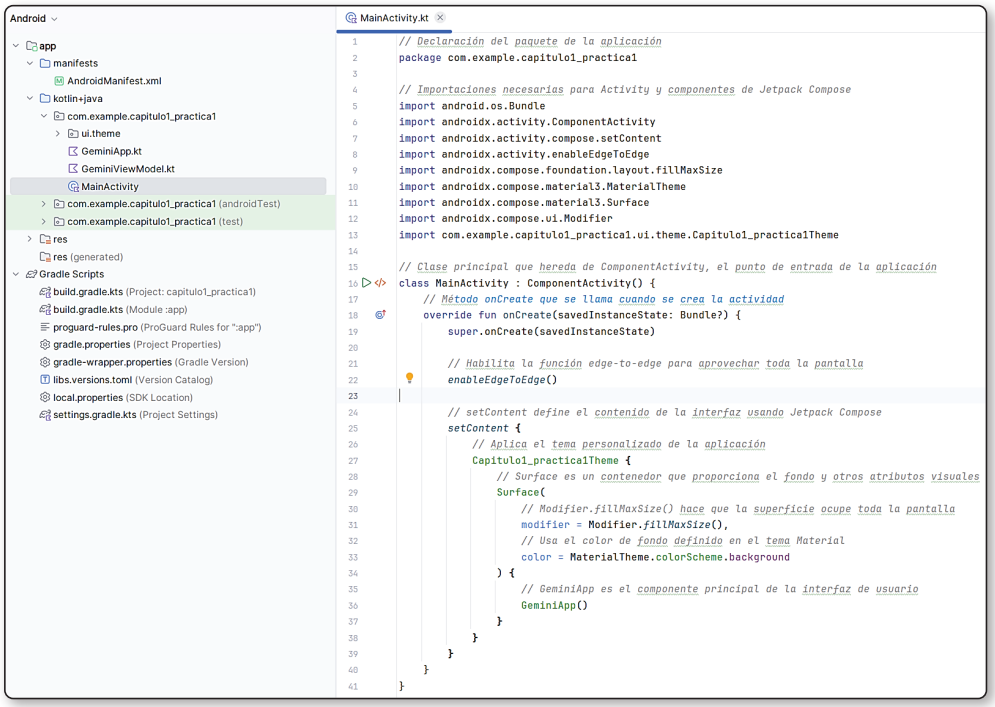
La función onCreate es especialmente importante ya que configura varios aspectos clave:

1. Habilita la visualización edge-to-edge, que permite que la aplicación utilice toda la pantalla del dispositivo, incluyendo las áreas alrededor de la cámara frontal o los bordes curvos.

2. Establece el contenido de la interfaz usando Compose mediante setContent, que configura una jerarquía de composables que incluye:

   - El tema personalizado de la aplicación.
   - Una superficie que ocupa toda la pantalla y proporciona el color de fondo.
   - El componente principal GeminiApp que contiene la lógica de la interfaz.

La estructura implementa el patrón de diseño Material Design 3 a través de MaterialTheme, asegurando una apariencia moderna y consistente en toda la aplicación.



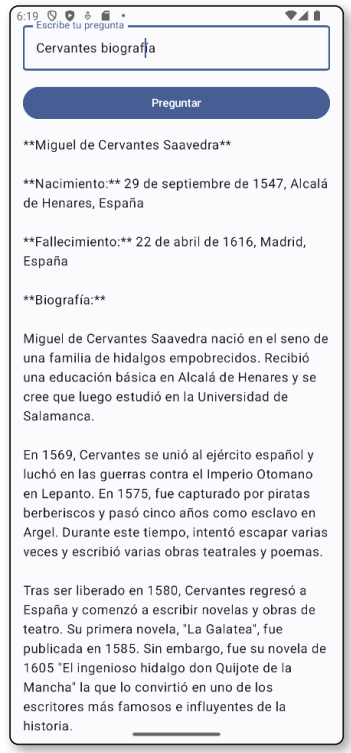
1_10

MainActivity.kt:

```
// Declaración del paquete de la aplicación
package com.example.capitulo1_practica1

// Importaciones necesarias para Activity y componentes de Jetpack Compose
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.ui.Modifier
import com.example.capitulo1_practica1.ui.theme.Capitulo1_practica1Theme

// Clase principal que hereda de ComponentActivity, el punto de entrada de la
aplicación
class MainActivity : ComponentActivity() {
    // Método onCreate que se llama cuando se crea la actividad
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Habilita la función edge-to-edge para aprovechar toda la pantalla
```

```
        enableEdgeToEdge()

        // setContent define el contenido de la interfaz usando Jetpack Compose
        setContent {
            // Aplica el tema personalizado de la aplicación
            Capitulo1_practica1Theme {
                // Surface es un contenedor que proporciona el fondo y otros
atributos visuales
                Surface(
                    // Modifier.fillMaxSize() hace que la superficie ocupe toda la
pantalla
                    modifier = Modifier.fillMaxSize(),
                    // Usa el color de fondo definido en el tema Material
                    color = MaterialTheme.colorScheme.background
                ) {
                    // GeminiApp es el componente principal de la interfaz de
usuario
                    GeminiApp()
                }
            }
        }
    }
}
```

Paso 6: imagen de la aplicación funcionando



1_11

1.4 BIBLIOGRAFÍA

[1] *https://es.wikipedia.org/wiki/Red_neuronal_artificial*

[2] *https://es.wikipedia.org/wiki/Transformador_(modelo_de_aprendizaje_autom%C3%A1tico)*

[3] *https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)*

[4] *https://en.wikipedia.org/wiki/Large_language_model*

[5] *https://en.wikipedia.org/wiki/Gemini_(language_model)*

[6] *https://en.wikipedia.org/wiki/Multimodal_learning*