**Capítulo 3 – Pytorch**

**convert.py**

import torch

import torchvision.models as models

# Load pre-trained ResNet18 model

model = models.resnet18(pretrained=True)

model.eval()

# Create input example (standard size for ResNet18)

example = torch.rand(1, 3, 224, 224)

# Export the model in mobile format

traced\_script\_module = torch.jit.trace(model, example)

traced\_script\_module.save("resnet18.pt")

print("ResNet18 model converted and saved as 'resnet18.pt'")

En **build.gradle.kts(Module :app)**:

plugins **{** alias(*libs*.*plugins*.*android*.*application*)  
 alias(*libs*.*plugins*.*kotlin*.*android*)  
 alias(*libs*.*plugins*.*kotlin*.*compose*)  
**}***android* **{** namespace = "com.example.capitulo3\_practica1"  
 compileSdk = 35  
  
 defaultConfig **{** applicationId = "com.example.capitulo3\_practica1"  
 minSdk = 34  
 targetSdk = 35  
 versionCode = 1  
 versionName = "1.0"  
  
 testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"  
 **}** buildTypes **{** *release* **{** isMinifyEnabled = false  
 proguardFiles(  
 getDefaultProguardFile("proguard-android-optimize.txt"),  
 "proguard-rules.pro"  
 )  
 **}  
 }** compileOptions **{** sourceCompatibility = JavaVersion.*VERSION\_11* targetCompatibility = JavaVersion.*VERSION\_11* **}** *kotlinOptions* **{** jvmTarget = "11"  
 **}** buildFeatures **{** compose = true  
 **}  
}***dependencies* **{** *implementation*(*libs*.*androidx*.*core*.*ktx*)  
 *implementation*(*libs*.*androidx*.*lifecycle*.*runtime*.*ktx*)  
 *implementation*(*libs*.*androidx*.*activity*.*compose*)  
 *implementation*(platform(*libs*.*androidx*.*compose*.*bom*))  
 *implementation*(*libs*.*androidx*.*ui*)  
 *implementation*(*libs*.*androidx*.*ui*.*graphics*)  
 *implementation*(*libs*.*androidx*.*ui*.*tooling*.*preview*)  
 *implementation*(*libs*.*androidx*.*material3*)  
 *testImplementation*(*libs*.*junit*)  
 *androidTestImplementation*(*libs*.*androidx*.*junit*)  
 *androidTestImplementation*(*libs*.*androidx*.*espresso*.*core*)  
 *androidTestImplementation*(platform(*libs*.*androidx*.*compose*.*bom*))  
 *androidTestImplementation*(*libs*.*androidx*.*ui*.*test*.*junit4*)  
 *debugImplementation*(*libs*.*androidx*.*ui*.*tooling*)  
 *debugImplementation*(*libs*.*androidx*.*ui*.*test*.*manifest*)  
  
 *// CameraX  
 implementation*("androidx.camera:camera-camera2:1.4.1")  
 *implementation*("androidx.camera:camera-lifecycle:1.4.1")  
 *implementation*("androidx.camera:camera-view:1.4.1")  
  
 *// Manejo de permisos  
 implementation*("com.google.accompanist:accompanist-permissions:0.32.0")  
  
 *// PyTorch Mobile  
 implementation*("org.pytorch:pytorch\_android:2.1.0")  
 *implementation*("org.pytorch:pytorch\_android\_torchvision:2.1.0")  
  
 *// ViewModel  
 implementation*("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")  
**}**

Preparación de **AndroidManifest.xml**:

**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">  
  
 *<!-- Permiso de cámara -->* <uses-permission android:name="android.permission.CAMERA" />  
  
 *<!-- Requisitos de funciones -->* <uses-feature android:name="android.hardware.camera" />  
 <uses-feature android:name="android.hardware.camera.autofocus" />  
  
 <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.Capitulo3\_practica1"  
 tools:targetApi="31">  
 <activity  
 android:name=".MainActivity"  
 android:exported="true"  
 android:label="@string/app\_name"  
 android:theme="@style/Theme.Capitulo3\_practica1">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN" />  
  
 <category android:name="android.intent.category.LAUNCHER" />  
 </intent-filter>  
 </activity>  
 </application>  
  
</manifest>

**Type.kt**:

Código de **Type.kt**:

package com.example.capitulo3\_practica1.ui.theme  
  
import androidx.compose.material3.Typography  
import androidx.compose.ui.text.TextStyle  
import androidx.compose.ui.text.font.FontFamily  
import androidx.compose.ui.text.font.FontWeight  
import androidx.compose.ui.unit.sp  
  
*// Conjunto de estilos de tipografía de Material para empezar*val *Typography* = Typography(  
 bodyLarge = TextStyle(  
 fontFamily = FontFamily.Default,  
 fontWeight = FontWeight.Normal,  
 fontSize = 16.*sp*,  
 lineHeight = 24.*sp*,  
 letterSpacing = 0.5.*sp* )  
 */\* Otros estilos de texto predeterminados para anular  
 titleLarge = TextStyle(  
 fontFamily = FontFamily.Default,  
 fontWeight = FontWeight.Normal,  
 fontSize = 22.sp,  
 lineHeight = 28.sp,  
 letterSpacing = 0.sp  
 ),  
 labelSmall = TextStyle(  
 fontFamily = FontFamily.Default,  
 fontWeight = FontWeight.Medium,  
 fontSize = 11.sp,  
 lineHeight = 16.sp,  
 letterSpacing = 0.5.sp  
 )  
 \*/*)

Código de **Color.kt**:

package com.example.capitulo3\_practica1.ui.theme  
  
import androidx.compose.ui.graphics.Color  
  
val *Purple80* = *Color*(0xFFD0BCFF)  
val *PurpleGrey80* = *Color*(0xFFCCC2DC)  
val *Pink80* = *Color*(0xFFEFB8C8)  
  
val *Purple40* = *Color*(0xFF6650a4)  
val *PurpleGrey40* = *Color*(0xFF625b71)  
val *Pink40* = *Color*(0xFF7D5260)

Código de **Theme.kt**:

package com.example.capitulo3\_practica1.ui.theme  
  
import android.app.Activity  
import androidx.compose.foundation.isSystemInDarkTheme  
import androidx.compose.material3.MaterialTheme  
import androidx.compose.material3.darkColorScheme  
import androidx.compose.material3.lightColorScheme  
import androidx.compose.runtime.Composable  
import androidx.compose.runtime.SideEffect  
import androidx.compose.ui.graphics.Color  
import androidx.compose.ui.platform.*LocalView*import androidx.core.view.WindowCompat  
  
private val *DarkColorScheme* = *darkColorScheme*(  
 primary = *Color*(0xFF6200EE),  
 secondary = *Color*(0xFF03DAC5),  
 tertiary = *Color*(0xFF3700B3),  
 background = Color.Black  
)  
  
private val *LightColorScheme* = *lightColorScheme*(  
 primary = *Color*(0xFF6200EE),  
 secondary = *Color*(0xFF03DAC5),  
 tertiary = *Color*(0xFF3700B3),  
 background = Color.White  
)  
  
@Composable  
fun Capitulo3\_practica1Theme(  
 darkTheme: Boolean = isSystemInDarkTheme(),  
 content: @Composable () -> Unit  
) {  
 val colorScheme = if (darkTheme) *DarkColorScheme* else *LightColorScheme* val view = *LocalView*.current  
 if (!view.*isInEditMode*) {  
 SideEffect **{** val window = (view.*context* as Activity).*window* val insetsController = WindowCompat.getInsetsController(window, view)  
  
 *// Configurar la apariencia de la barra de estado según el tema* insetsController.*isAppearanceLightStatusBars* = !darkTheme  
 **}** }  
  
 MaterialTheme(  
 colorScheme = colorScheme,  
 content = content  
 )  
}

Código de **ImageClassifier.kt**:

package com.example.capitulo3\_practica1  
  
import android.content.Context  
import android.graphics.Bitmap  
import android.util.Log  
import org.pytorch.IValue  
import org.pytorch.Module  
import org.pytorch.Tensor  
import org.pytorch.torchvision.TensorImageUtils  
import java.io.File  
import java.io.FileOutputStream  
import java.io.IOException  
  
private const val *TAG* = "ImageClassifier"  
private const val *MODEL\_NAME* = "resnet18.pt"  
private const val *CLASSES\_FILE\_NAME* = "imagenet\_classes.txt"  
  
class ImageClassifier(private val context: Context) {  
  
 private var module: Module? = null  
 private var classes: List<String> = *emptyList*()  
  
 init {  
 try {  
 *// Cargar el modelo de PyTorch* module = Module.load(assetFilePath(context, *MODEL\_NAME*))  
  
 *// Cargar las clases para ImageNet* classes = loadClasses()  
  
 Log.d(*TAG*, "Modelo y clases cargadas exitosamente")  
 } catch (e: IOException) {  
 Log.e(*TAG*, "Error al cargar el modelo o las clases", e)  
 }  
 }  
  
 fun classify(bitmap: Bitmap): String {  
 if (module == null) {  
 return "Modelo no cargado"  
 }  
  
 try {  
 *// Preprocesar la imagen (normalizar utilizando la media y estándar de ImageNet)* val inputTensor = TensorImageUtils.bitmapToFloat32Tensor(  
 bitmap,  
 TensorImageUtils.*TORCHVISION\_NORM\_MEAN\_RGB*,  
 TensorImageUtils.*TORCHVISION\_NORM\_STD\_RGB* )  
  
 *// Ejecutar el modelo* val outputTensor = module!!.forward(IValue.from(inputTensor)).toTensor()  
  
 *// Obtener la clase predecida* val scores = outputTensor.*dataAsFloatArray  
  
 // Encuentrar el índice con puntuación máxima* var maxScore = Float.NEGATIVE\_INFINITY  
 var maxScoreIdx = -1  
  
 for (i in scores.*indices*) {  
 if (scores[i] > maxScore) {  
 maxScore = scores[i]  
 maxScoreIdx = i  
 }  
 }  
  
 val className = if (maxScoreIdx >= 0 && maxScoreIdx < classes.size) {  
 classes[maxScoreIdx]  
 } else {  
 "Unknown"  
 }  
  
 val confidence = if (maxScore > 0) {  
 *// Convertir a porcentaje* String.*format*("%.1f%%", maxScore \* 100)  
 } else {  
 "low"  
 }  
  
 return "$className ($confidence)"  
 } catch (e: Exception) {  
 Log.e(*TAG*, "Error durante la clasificación de imágenes", e)  
 return "Classification error"  
 }  
 }  
  
 private fun loadClasses(): List<String> {  
 return try {  
 context.*assets*.open(*CLASSES\_FILE\_NAME*).*bufferedReader*().*readLines*()  
 } catch (e: IOException) {  
 Log.e(*TAG*, "Error al cargar clases", e)  
 *emptyList*()  
 }  
 }  
  
 fun close() {  
 module = null  
 }  
  
 @Throws(IOException::class)  
 private fun assetFilePath(context: Context, assetName: String): String {  
 val file = File(context.*filesDir*, assetName)  
  
 if (file.exists() && file.length() > 0) {  
 return file.*absolutePath* }  
  
 context.*assets*.open(assetName).*use* **{** inputStream **->** FileOutputStream(file).*use* **{** outputStream **->** val buffer = ByteArray(4 \* 1024)  
 var read: Int  
 while (inputStream.read(buffer).*also* **{** read = **it }** != -1) {  
 outputStream.write(buffer, 0, read)  
 }  
 outputStream.flush()  
 **}  
 }** return file.*absolutePath* }  
}

Código de **CameraPreview.kt**:

package com.example.capitulo3\_practica1  
  
import android.content.Context  
import android.graphics.Bitmap  
import android.graphics.Matrix  
import android.util.Log  
import androidx.camera.core.CameraSelector  
import androidx.camera.core.ImageAnalysis  
import androidx.camera.core.ImageCapture  
import androidx.camera.core.ImageProxy  
import androidx.camera.core.Preview  
import androidx.camera.lifecycle.ProcessCameraProvider  
import androidx.camera.view.PreviewView  
import androidx.compose.foundation.layout.fillMaxSize  
import androidx.compose.runtime.Composable  
import androidx.compose.runtime.DisposableEffect  
import androidx.compose.runtime.remember  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.platform.*LocalContext*import androidx.compose.ui.viewinterop.AndroidView  
import androidx.core.content.ContextCompat  
import androidx.lifecycle.Lifecycle  
import androidx.lifecycle.LifecycleOwner  
import java.util.concurrent.Executors  
import kotlin.coroutines.resume  
import kotlin.coroutines.suspendCoroutine  
  
private const val *TAG* = "CameraPreview"  
  
@Composable  
fun CameraPreview(  
 onFrameAnalyzed: (Bitmap) -> Unit  
) {  
 val context = *LocalContext*.current  
  
 *// Cree un propietario de ciclo de vida con el alcance adecuado* val lifecycleOwner = rememberLifecycleOwner()  
  
 val preview = Preview.Builder().build()  
 val previewView = remember **{** PreviewView(context) **}** val imageCapture = remember **{** ImageCapture.Builder().build() **}** val cameraSelector = remember **{** CameraSelector.*DEFAULT\_BACK\_CAMERA* **}** val imageAnalyzer = ImageAnalysis.Builder()  
 .setBackpressureStrategy(ImageAnalysis.*STRATEGY\_KEEP\_ONLY\_LATEST*)  
 .build()  
 .*apply* **{** setAnalyzer(Executors.newSingleThreadExecutor(), ImageAnalyzer **{** bitmap **->** onFrameAnalyzed(bitmap)  
 **}**)  
 **}** *// Usar DisposableEffect* DisposableEffect(lifecycleOwner) **{** val cameraProviderFuture = ProcessCameraProvider.getInstance(context)  
 val cameraProvider = cameraProviderFuture.get()  
  
 try {  
 *// Desvincular todos los casos de uso antes de volver a vincularlos* cameraProvider.unbindAll()  
 *// Vincular casos de uso a la cámara* cameraProvider.bindToLifecycle(  
 lifecycleOwner,  
 cameraSelector,  
 preview,  
 imageCapture,  
 imageAnalyzer  
 )  
  
 preview.setSurfaceProvider(previewView.*surfaceProvider*)  
 } catch (e: Exception) {  
 Log.e(*TAG*, "Error en la vinculación del caso de uso", e)  
 }  
  
 onDispose **{** cameraProvider.unbindAll()  
 **}  
 }** AndroidView(  
 factory = **{** previewView **}**,  
 modifier = Modifier.*fillMaxSize*()  
 )  
}  
  
*/\*\*  
 \* Crea un propietario de ciclo de vida que sigue el ciclo de vida de la composición.  
 \*/*@Composable  
private fun rememberLifecycleOwner(): LifecycleOwner {  
 val context = *LocalContext*.current  
  
 return remember **{** ComposeLifecycleOwner()  
 **}**.*apply* **{** this.registerLifecycle()  
 **}**}  
  
*/\*\*  
 \* Implementación de LifecycleOwner personalizada que se puede utilizar dentro de Compose.  
 \*/*private class ComposeLifecycleOwner : LifecycleOwner {  
 private val lifecycleRegistry = androidx.lifecycle.LifecycleRegistry(this)  
  
 override val lifecycle: Lifecycle  
 get() = lifecycleRegistry  
  
 fun registerLifecycle() {  
 lifecycleRegistry.currentState = Lifecycle.State.*RESUMED* }  
  
 fun unregisterLifecycle() {  
 lifecycleRegistry.currentState = Lifecycle.State.*DESTROYED* }  
}  
  
private class ImageAnalyzer(private val onImageAnalyzed: (Bitmap) -> Unit) : ImageAnalysis.Analyzer {  
 override fun analyze(imageProxy: ImageProxy) {  
 val bitmap = imageProxy.toBitmap()  
  
 *// Asegurar de que la rotación y el análisis de las imágenes sean adecuados* val rotatedBitmap = bitmap?.*let* **{** rotateBitmap(**it**, imageProxy.*imageInfo*.*rotationDegrees*.toFloat())  
 **}** rotatedBitmap?.*let* **{** *// Cambiar el tamaño del mapa de bits para el modelo (tamaños comunes como 224x224 o 299x299)* val resizedBitmap = Bitmap.createScaledBitmap(**it**, 224, 224, true)  
 onImageAnalyzed(resizedBitmap)  
 **}** imageProxy.close()  
 }  
  
 private fun ImageProxy.toBitmap(): Bitmap? {  
 val planeProxy = *planes*[0]  
 val buffer = planeProxy.*buffer* val bytes = ByteArray(buffer.remaining())  
 buffer.get(bytes)  
  
 return android.graphics.BitmapFactory.decodeByteArray(bytes, 0, bytes.size)  
 }  
  
 private fun rotateBitmap(bitmap: Bitmap, rotationDegrees: Float): Bitmap {  
 val matrix = Matrix()  
 matrix.postRotate(rotationDegrees)  
 return Bitmap.createBitmap(bitmap, 0, 0, bitmap.*width*, bitmap.*height*, matrix, true)  
 }  
}  
  
suspend fun Context.getCameraProvider(): ProcessCameraProvider = suspendCoroutine **{** continuation **->** ProcessCameraProvider.getInstance(this).*also* **{** cameraProvider **->** cameraProvider.addListener(**{** continuation.*resume*(cameraProvider.get())  
 **}**, ContextCompat.getMainExecutor(this))  
 **}  
}**

Código de **CameraViewModel.kt**:

package com.example.capitulo3\_practica1  
  
import android.app.Application  
import android.graphics.Bitmap  
import androidx.compose.runtime.State  
import androidx.compose.runtime.mutableStateOf  
import androidx.lifecycle.AndroidViewModel  
import androidx.lifecycle.*viewModelScope*import kotlinx.coroutines.Dispatchers  
import kotlinx.coroutines.launch  
import kotlinx.coroutines.withContext  
  
class CameraViewModel(application: Application) : AndroidViewModel(application) {  
  
 private val classifier = ImageClassifier(application)  
  
 private val \_classificationResult = *mutableStateOf*("Analizando...")  
 val classificationResult: State<String> = \_classificationResult  
  
 fun classifyImage(bitmap: Bitmap) {  
 *viewModelScope*.*launch* **{** val result = withContext(Dispatchers.Default) **{** classifier.classify(bitmap)  
 **}** \_classificationResult.value = result  
 **}** }  
  
 override fun onCleared() {  
 super.onCleared()  
 classifier.close()  
 }  
}

Código de **MainActivity.kt**:

package com.example.capitulo3\_practica1  
  
import android.Manifest  
import android.os.Bundle  
import androidx.activity.ComponentActivity  
import androidx.activity.compose.setContent  
import androidx.compose.foundation.layout.Box  
import androidx.compose.foundation.layout.fillMaxSize  
import androidx.compose.foundation.layout.padding  
import androidx.compose.material3.MaterialTheme  
import androidx.compose.material3.Surface  
import androidx.compose.material3.Text  
import androidx.compose.runtime.Composable  
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.graphics.Color  
import androidx.compose.ui.text.font.FontWeight  
import androidx.compose.ui.unit.dp  
import androidx.compose.ui.unit.sp  
import androidx.lifecycle.viewmodel.compose.viewModel  
import com.example.capitulo3\_practica1.ui.theme.Capitulo3\_practica1Theme  
import com.google.accompanist.permissions.ExperimentalPermissionsApi  
import com.google.accompanist.permissions.*isGranted*import com.google.accompanist.permissions.rememberPermissionState  
import com.google.accompanist.permissions.*shouldShowRationale*class MainActivity : ComponentActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 *setContent* **{** Capitulo3\_practica1Theme **{** Surface(  
 modifier = Modifier.*fillMaxSize*(),  
 color = MaterialTheme.colorScheme.background  
 ) **{** CameraClassifierApp()  
 **}  
 }  
 }** }  
}  
  
@OptIn(ExperimentalPermissionsApi::class)  
@Composable  
fun CameraClassifierApp(  
 cameraViewModel: CameraViewModel = viewModel()  
) {  
 val cameraPermissionState = rememberPermissionState(Manifest.permission.*CAMERA*)  
  
 if (cameraPermissionState.status.*isGranted*) {  
 *// Permission is granted, show camera and classifier  
 // Se concede permiso, mostrar cámara y clasificador.* Box(modifier = Modifier.*fillMaxSize*()) **{** *// Camera Preview* CameraPreview(  
 onFrameAnalyzed = **{** bitmap **->** cameraViewModel.classifyImage(bitmap)  
 **}** )  
  
 *// Classification Results  
 // Resultados de la clasificación* val classificationResult by cameraViewModel.classificationResult  
  
 Text(  
 text = classificationResult,  
 color = Color.White,  
 fontSize = 22.*sp*,  
 fontWeight = FontWeight.Bold,  
 modifier = Modifier  
 .*align*(Alignment.BottomCenter)  
 .*padding*(bottom = 32.*dp*)  
 .*padding*(horizontal = 16.*dp*)  
 )  
 **}** } else {  
 *// Request camera permission  
 // Solicitar permiso para la cámara* PermissionRequest(  
 permissionState = cameraPermissionState,  
 shouldShowRationale = cameraPermissionState.status.*shouldShowRationale*,  
 onPermissionResult = **{** isGranted **->** if (isGranted) {  
 *// Permission granted  
 // Permiso concedido* }  
 **}** )  
 }  
}  
  
@OptIn(ExperimentalPermissionsApi::class)  
@Composable  
fun PermissionRequest(  
 permissionState: com.google.accompanist.permissions.PermissionState,  
 shouldShowRationale: Boolean,  
 onPermissionResult: (Boolean) -> Unit  
) {  
 Box(  
 modifier = Modifier.*fillMaxSize*(),  
 contentAlignment = Alignment.Center  
 ) **{** var requestedPermission by remember **{** *mutableStateOf*(false) **}** if (!requestedPermission) {  
 Text(  
 text = if (shouldShowRationale) {  
 "Se necesita permiso de la cámara para clasificar las imágenes."  
 } else {  
 "Por favor, conceda permiso a la cámara para usar esta aplicación."  
 },  
 modifier = Modifier.*padding*(bottom = 24.*dp*), *// Aumentar el espacio inferior* textAlign = androidx.compose.ui.text.style.TextAlign.Center *// Centrar el texto* )  
  
 androidx.compose.material3.Button(  
 onClick = **{** requestedPermission = true  
 permissionState.launchPermissionRequest()  
 **}**,  
 modifier = Modifier.*padding*(top = 16.*dp*)  
 ) **{** Text("Request Permission")  
 **}** }  
 **}**}