Intégration de CameraX dans une Application Android en Kotlin

Ce guide explique comment intégrer et configurer **CameraX** pour capturer un flux vidéo en temps réel dans une application Android Kotlin.

Prérequis

Avant de commencer, assurez-vous de disposer des éléments suivants :

- **Android Studio** (version Arctic Fox ou plus récente recommandée)
- **Kotlin** configuré comme langage principal
- Un appareil Android ou un émulateur prenant en charge CameraX
- Permissions configurées pour accéder à la caméra de l'appareil

---

Étape 1 : Configuration du Projet

1. **Ajoutez les dépendances CameraX dans le fichier `build.gradle` :**

   Dans le fichier `app/build.gradle`, ajoutez les lignes suivantes :

   ```
   implementation(libs.androidx.camera.core)
   implementation(libs.androidx.camera.camera2)
   implementation(libs.androidx.camera.lifecycle)
   implementation(libs.androidx.camera.video)
   implementation(libs.androidx.camera.view)
   implementation(libs.androidx.camera.extensions)
   ```

2. **Ajoutez les permissions nécessaires dans AndroidManifest.xml :**

   ```
   <uses-feature android:name="android.hardware.camera.any" />
   <uses-permission android:name="android.permission.CAMERA" />
   <uses-permission android:name="android.permission.RECORD_AUDIO" />
   <uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"
       android:maxSdkVersion="28" />
   ```

   Cela garantit que l'application peut utiliser la caméra.

Étape 2 : Création de l'Interface Utilisateur

Ajoutez un PreviewView et un TextView dans votre fichier de mise en page :

Dans `activity_main.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.camera.view.PreviewView
        android:id="@+id/viewFinder"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <TextView
        android:id="@+id/predictedDigit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="24sp"
        android:textColor="@android:color/black"
        android:layout_gravity="center_horizontal"
        android:padding="16dp"
        android:text="Predicted Digit: -" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Étape 3 : Configuration de CameraX dans Kotlin

Créez une classe principale `MainActivity` :

Voici un exemple complet de configuration de CameraX :

```
import android.Manifest
import android.content.Context
import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.camera.lifecycle.ProcessCameraProvider
import androidx.camera.view.PreviewView
import androidx.core.content.ContextCompat
import com.example.myapplication.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
    private lateinit var viewBinding: ActivityMainBinding
    private lateinit var cameraController: LifecycleCameraController
    private lateinit var predictedDigitTextView: TextView
```

```
        override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewBinding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(viewBinding.root)

        viewModel.predictedDigit.observe(this, Observer { digit ->
            viewBinding.predictedDigit.text = digit
        })

        if (!hasPermissions(baseContext)) {
            activityResultLauncher.launch(REQUIRED_PERMISSIONS)
        } else {
            startCamera()
        }
    }

    private fun startCamera() {
        val previewView: PreviewView = viewBinding.viewFinder
        cameraController = LifecycleCameraController(baseContext)
        cameraController.bindToLifecycle(this)
        previewView.controller = cameraController
    }

    private val activityResultLauncher =

registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions())
{ permissions ->
            var permissionGranted = true
            permissions.entries.forEach {
                if (it.key in REQUIRED_PERMISSIONS && it.value == false) {
                    permissionGranted = false
                }
                if (!permissionGranted) {
                    Toast.makeText(this, "Permission request denied.",
Toast.LENGTH_LONG).show()
                } else {
                    startCamera()
                }
            }
        }
    companion object {
        private const val TAG = "MaNumber"
        private const val FILENAME_FORMAT = "yyyy-MM-dd-HH-mm-ss-SSS"
        private val REQUIRED_PERMISSIONS =
            mutableListOf(
                Manifest.permission.CAMERA
            ).apply {
                if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.P) {
                    add(Manifest.permission.WRITE_EXTERNAL_STORAGE)
                }
            }.toTypedArray()
```

```
        fun hasPermissions(context: Context) = REQUIRED_PERMISSIONS.all {
            ContextCompat.checkSelfPermission(context, it) ==
PackageManager.PERMISSION_GRANTED
        }
    }
}
```

Étape 4 : Tester l'Application

Exécutez votre projet sur un appareil ou un émulateur Android :

- Le flux vidéo en direct devrait s'afficher dans le PreviewView.

Résolution des problèmes éventuels :

- Si le flux vidéo ne s'affiche pas, assurez-vous que les permissions caméra ont été accordées.
- Vérifiez la compatibilité de l'appareil avec CameraX.

Étape 5 : Étapes Supplémentaires

- Ajouter une gestion avancée des permissions :
  Utilisez ActivityResultContracts pour gérer les permissions de manière moderne.
- Intégrer un modèle de prédiction (ex. SVM) :
  Traitez les images capturées en temps réel et affichez les résultats.

---

image processing

Le fichier Image_Processing.kt contient une série de fonctions en Kotlin permettant de réaliser différentes opérations de traitement d'images, telles que le filtrage, la détection de contours, le redimensionnement et la binarisation. Ces opérations sont essentielles pour préparer les images récupérées par CameraX de l'application mobile. Une fois préparées, ces images sont transformées en vecteurs pour être transmises au modèle d'apprentissage automatique afin d'effectuer des prédictions.

loadImage(imagePath: String): BufferedImage
Description : Charge une image depuis le chemin spécifié.
Utilisation :
val image = loadImage("res/images/test0.jpeg")

applyGaussianBlur(image: BufferedImage): BufferedImage
Description : Applique un filtre gaussien pour réduire le bruit de l'image.
Détails : Utilise une matrice de convolution 5x5 pour le lissage.
Utilisation :
val blurredImage = applyGaussianBlur(image)

applySobelEdgeDetection(image: BufferedImage): BufferedImage
Description : Applique le filtre Sobel pour détecter les contours horizontaux et verticaux.

Remarque : Dans ce projet, nous n'appliquons pas le filtre Sobel car nous avons constaté que les chiffres n'étaient pas bien récupérés après cette étape de détection des contours. Le lissage par filtre gaussien suivi de la binarisation donne de meilleurs résultats pour notre cas d'usage.
Utilisation :
val edgeDetectedImage = applySobelEdgeDetection(image)

resizeImage(image: BufferedImage, width: Int, height: Int): BufferedImage
Description : Redimensionne l'image aux dimensions spécifiées.
Utilisation :
val resizedImage = resizeImage(image, 28, 28)

otsuThreshold(image: BufferedImage): Int
Description : Calcule le seuil optimal de binarisation en utilisant la méthode d'Otsu.
Utilisation :
val threshold = otsuThreshold(image)

binarizeImage(image: BufferedImage, threshold: Int): Array
Description : Binarise l'image en utilisant le seuil spécifié.
Utilisation :
val binarizedPixels = binarizeImage(image, threshold)

flattenBinarizedImage(binarizedPixels: Array): IntArray
Description : Aplatie l'image binarisée en un vecteur unidimensionnel.
Utilisation :
val flattenedVector = flattenBinarizedImage(binarizedPixels)

GADEU MONTHE VINETTE MARCY

Implémentation du SVM avec le noyau RBF en Java

Après avoir programmé notre SVM en Python à l'aide de scikit-learn , nous avons récupéré les vecteurs de support, les coefficients et les biais de chaque classe soit 10 classes (de 0 à 9).

Nous avons implémenté la fonction de décision SVM suivante :
$$
f(x) = \text{sign} \left(\sum_{i=1}^n \alpha_i y_i K(x_i, x) + b \right)
$$

Où :

- (x) : le vecteur image.
- (x_i) : les vecteurs supports.
- (\alpha_i) : les coefficients associés aux vecteurs supports.
- (b) : le biais appris.
- (\gamma) : le paramètre du noyau.

Le noyau utilisé dans cet algorithme est le noyau gaussien (RBF), défini par :
$$ K(x_i, x) = \exp \left(-\gamma |x_i - x|^2 \right)

$$

Nous prédisons les scores d'appartenance de notre iamge à chaque classe et la classe avec le score maximale est la classe prédite.

Pour ce faire, nous avons organisé notre code comme suite:

1. **Classe** `ModeleSVM`

cette classe représente le modèle SVM avec un noyau RBF.

Attributs principaux :

- `vecteursSupport` : Matrice contenant les vecteurs de support de la classe.
- `coefficients` : Coefficients (\alpha_i) associés aux vecteurs de support.
- `biais` : Le biais (b).
- `gamma` : Paramètre du noyau RBF (défini à 0.001 par défaut).

Méthodes principales :

1. `ModeleSVM(String fichierSupportVecteurs, String fichierCoefficients, String fichierBiais)`
   - Constructeur qui initialise le modèle en chargeant les paramètres depuis les fichiers en spécifiant le chemin des fichiers.
2. `chargerVecteursSupports(String fichierSupportVecteurs)`
   - Charge les vecteurs supports depuis un fichier.
3. `chargerCoefficients(String fichierCoefficients)`
   - Charge les coefficients (\alpha_i) depuis un fichier.
4. `chargerBiais(String fichierBiais)`
   - Charge le biais (b) depuis un fichier.
5. `noyau_rbf(double[] vecteur_image, double[] vecteurs_support)`
   - Calcule la valeur du noyau RBF entre un vecteur d'entrée et un vecteur support.
6. `predict_score(double[] vecteur_image)`
   - Calcule le score pour une classe donnée en fonction du vecteur image.

2. **Classe** `ListModeleSVM`

Elle contient la liste des 10 modèles SVM des 10 classes.

Attributs principaux :

- `listesDeModeleSVM` : Une liste contenant 10 instances de `ModeleSVM`.

Méthodes principales :

1. `ListModeleSVM()`
   - Constructeur qui initialise la liste en chargeant les paramètres de chaque modèle depuis les fichiers correspondants.

2. `predict_chiffre(double[] vecteur_image)`
    - Prédit la classe d'un vecteur d'entrée en calculant les scores pour les 10 classes et retourne la classe avec le score maximal.

Après avoir écrit le modèle en Java, nous l'avons traduit en Kotlin pour pouvoir l'intégrer facilement dans l'application.

Références

- Documentation officielle de CameraX
- Guide Android Developers

Auteur

Lotfi Abdelkadir RABAH

Projet réalisé dans le cadre de MaNumber.