Tecnológico Nacional de México

Campus Culiacán

Carrera:

Ingeniería en Sistemas Computacionales

Nombre de la materia:

Inteligencia Artificial

Tarea 2: Sistema detector de emociones

Alumnos:

Aguilar Recio Jesús Octavio

Flores Fernandez Emily Karely

Nombre del maestro:

Zuriel Dathan Mora Felix

Grupo:

9:00 – 10:00

Introducción

Esta tarea es la continuación de la tarea de preprocesamiento. Para poder hacer un sistema para detectar emociones por medio de la cámara de la computadora en tiempo real, se necesita seguir una serie de pasos, los cuales comenzamos con la obtención del conjunto de imágenes y preprocesarlas para una detección más precisa. Los siguientes pasos consisten en seleccionar la arquitectura neuronal, después definir los parámetros de entrenamiento y por último realizar pruebas con el modelo utilizando la cámara de la computadora. Es por eso por lo que en este documento se explicará la elaboración de cada paso y los resultados obtenidos.

Arquitectura de la red neuronal

La arquitectura escogida para este proyecto fue una red neuronal convolucional (CNN) la cual consta de una capa de entrada, una capa de salida y varias capas ocultas entre ambas, en las que dichas capas realizan operaciones que modifican los datos, con el propósito de comprender sus características particulares. Las capas comunes son: convolución, activación y agrupación.

En nuestro proyecto creamos un archivo llamado modelo.py el cual contiene la lógica de la arquitectura convolucional la cual esta especializada en reconocimiento de emociones faciales, ósea que sería el cerebro del proyecto el cual aprende los patrones de cada emoción.

Primero definimos la clase EmocionCNN y definimos la primera y segunda capa de convolución, en la cual la primera detecta patrones básicos como borde, cambios de intensidad utilizando escala de grises, 32 filtros de tamaño 3x3 píxeles y la segunda convolución detecta patrones más complejos usando 64 filtros.

```
# Capa convolucionales
self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
```

Después tenemos la capa de activación (relu) en la que solo se activa después de la convolución 1 y 2 esta permite al modelo aprender relaciones no lineales y en la capa de agrupación (pooling) se encarga de la reducción ósea que disminuye las dimensiones manteniendo solamente las características más importantes de la imagen.

```
# Primer bloque conv1 + relu + agrupar
x = F.relu(self.conv1(x))
x = F.max_pool2d(x, 2)  # 48x48 -> 24x24

# Segundo bloque conv2 + relu + agrupar
x = F.relu(self.conv2(x))
x = F.max_pool2d(x, 2)  # 24x24 -> 12x12

# Aplanar
x = x.view(-1, 64 * 12 * 12)
```

El fujo de la arquitectura sería entonces:

- Primera convolución (32 filtros 3x3) + relu + pooling: reduciendo la imagen de 48x48 a 24x24.
- Segunda convolución (64 filtros 3x3) + relu + pooling reduce ahora las imágenes a 12x12.
- Clasificación dos capas completamente conectadas 128 neuronas y 5 salidas = emociones.

Entrenamiento del modelo

En esta otra parte del código nos encargamos del entrenamiento de la red neuronal convolucional para pueda reconocer las emociones. En esta parte lo que hacemos es cargar y preparar los datos de entrenamiento, configurar el modelo CNN, entrenar la red y evaluar el rendimiento de esta.

Lo primero es la configuración inicial: detectar automáticamente si hay GPU disponible para acelerar el entrenamiento utilizando las librerías de torch. Y también obtenemos la dirección de las imágenes preprocesadas y cargamos los datos del archivo csv etiquetas_preprocesadas.

```
# configuracion del dispositivo
dispositivo = torch.device("cuda" if torch.cuda.is_available() else "cpu")
dir_datos = "D:\\Documentos\\Octavio\\TEC\\OCTAVO SEMESTRE\\Inteligencia A
datosCsv = os.path.join(dir_datos, "etiquetas_preprocesadas.csv")
```

Después al iniciar la clase emocionesDataset mapeamos los nombres de las carpetas de emociones con números id para cada una de ellas y aplicamos las transformaciones a las imágenes.

```
class emocionesDataset(Dataset):
    def __init__(self, datos_df, transformaciones=None):
        self.datos = datos_df
        self.transformaciones = transformaciones
        self.idEmociones = {"angry": 0, "distress": 1, "happy": 2, "sad": 3, "surprise": 4}
```

Creamos la función getitem para acceder a un elemento o elementos de un objeto utilizando los [ ] como una lista. De la dir_datos obtenemos las columnas del archivo csv, después leemos la imagen y redimensionamos la imagen.

```
def __getitem__(self, idx):
    rutImagen = os.path.join(dir_datos,
                             self.datos.iloc[idx]['conjunto'],
                             self.datos.iloc[idx]['etiqueta'],
                             self.datos.iloc[idx]['imagen'])

    img = cv2.imread(rutImagen, cv2.IMREAD_GRAYSCALE)
    etiqueta = self.idEmociones[self.datos.iloc[idx]['etiqueta']]

    if self.transformaciones:
        img = self.transformaciones(img)

    return img, etiqueta
```

Dentro de las transformaciones para entrenamiento y validación se realizan los siguientes procesos: convertimos a formato PIL y transformamos a tensor pytorch y normalizamos los valores a pixeles media 0.5 y desviación 0.5. Esto de las transformaciones se hace con el objetivo de mantener la consistencia en el formato de entrada.

```
# transformaciones para el datset
transformacionesTrain = transforms.Compose([
    transforms.ToPILImage(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

transformacionesVal = transforms.Compose([
    transforms.ToPILImage(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])
```

Dentro de la función principal main cargamos y preparamos los datos por ejemplo: obtenemos del dataset del csv para dividir aquellos que tengan la etiqueta train y val después les aplicamos las transformaciones para mantener la consistencia y por ultimo cargamos los dataloader para indicarle que procese simultáneamente 64 imágenes y que mezcle los datos para evitar sesgos.

```
# cargar el csv
datos = pd.read_csv(datosCsv)

# dividir el dataset en train y test
datosTrain = datos[datos['conjunto'] == 'train']
datosVal = datos[datos['conjunto'] == 'val']

# cargar el dataset y aplicar las transformaciones
datasetTrain = emocionesDataset(datosTrain, transformacionesTrain)
datasetVal = emocionesDataset(datosVal, transformacionesVal)

# cargar los dataLoaders para el entrenamiento y validacion
cargaTrain = DataLoader(datasetTrain, batch_size=64, shuffle=True)
cargaVal = DataLoader(datasetVal, batch_size=64, shuffle=False)
```

Después inicializamos el modelo llamando a la clase EmocionCNM. También inicializamos la función CrossEntropyLoss la cual calcula la perdida de las salidas de la red neuronal. Tambien inicializamos Adam el cual es un optimizador que adapta las tasas de aprendizaje ósea que minimiza la función de perdida durante el entrenamiento de la red.

```
modelo = EmocionCNN(num_classes=5).to(dispositivo)
criterio = nn.CrossEntropyLoss()
optimizador = optim.Adam(modelo.parameters(), lr=0.001)
```

Después entra el bucle de entrenamiento el cual le indicamos que de 30 pasadas completas por el dataset. Inicializamos la mejor perdida de validación y el modelo en modo entrenamiento. También reseteamos al inicio de cada vuelta los gradientes y con el forward pass (salida = modelo(entrada)) la imagen pasa por todas las capas conv + relu + pooling. Comparamos con crossentropy la predicción vs etiqueta real para calcular la perdida. Con backward calculamos los gradientes y con step actualizamos pesos.

```python
# entrenar el modelo
for epoch in range(30):
    # inicializar la mejor perdida de validacion
    modelo.train()
    perdidaTrain = 0.0

    for entrada, etiqueta in cargaTrain:
        entrada, etiqueta = entrada.to(dispositivo), etiqueta.to(dispositivo)
        #
        optimizador.zero_grad()
        salida = modelo(entrada)
        perdida = criterio(salida, etiqueta)
        perdida.backward()
        optimizador.step()

        perdidaTrain += perdida.item()
```

Seguido de esto evaluamos el modelo el cual desactiva dropout y batch norm para evaluación después hacemos el cálculo de métricas y al último vamos mostrando por la consola la muestra precisión, recall y F1-score.

```python
modelo.eval()
perdidaVal = 0.0
predicciones = []
etiquetas = []
# calcular las predicciones
with torch.no_grad():
    for entrada, etiqueta in cargaVal:
        entrada, etiqueta = entrada.to(dispositivo), etiqueta.to(dispositivo)
        salida = modelo(entrada)
        perdida = criterio(salida, etiqueta)
        perdidaVal += perdida.item()

        _, preds = torch.max(salida, 1)
        predicciones.extend(preds.cpu().numpy())
        etiquetas.extend(etiqueta.cpu().numpy())

# mostrar resultados
print(f'Epoch {epoch+1}/30')
print(f'Train Loss: {perdidaTrain/len(cargaTrain):.4f} | Val Loss: {perdidaVal/len(cargaVal):.4f}')
print(classification_report(etiquetas, predicciones, target_names=["angry", "distress", "happy", "sa
```

```
Epoch 30/30
Train Loss: 0.1570 | Val Loss: 2.6951
              precision    recall  f1-score   support

       angry       0.59      0.52      0.56       608
    distress       0.80      0.37      0.50        76
       happy       0.79      0.81      0.80      1098
         sad       0.59      0.63      0.61       715
    surprise       0.74      0.82      0.78       450

    accuracy                           0.70      2947
   macro avg       0.70      0.63      0.65      2947
weighted avg       0.70      0.70      0.69      2947
```

Por último en el entrenamiento guardamos el modelo solo aquel que su pérdida de validación sea menor a la mejor perdida o que el epoch sea igual a cero.

```
if epoch == 0 or perdidaVal < best_val_loss:
    best_val_loss = perdidaVal
    torch.save(modelo.state_dict(), 'modeloEmociones.pth')
```

Pruebas del modelo

Para poder probar el modelo, hicimos un sistema de reconocimiento de emociones en tiempo real el cual captura video de la cámara web, después detecta rostros de cada frame y clasifica la emoción y muestra el resultado en la pantalla.

Primero inicializamos el contructor de la clase DetectorEmociones, el cual configuramos primero el dispositivo GPU y cargamos el modelo pre-entrenado, preparamos las transformaciones de imagen que son iguales a las utilizadas en el entrenamiento, definimos las etiquetas de emociones. Y utilizando las librerías de cv2 inicializamos el detector de rostros hearcascade.

```
class DetectorEmociones:
    def __init__(self, rutModelo):
        self.dispositivo = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.modelo = self.cargarModelo(rutModelo)
        self.transformaciones = transforms.Compose([
            transforms.ToPILImage(),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.5], std=[0.5])
        ])
        self.emociones = ["enojado", "angustia", "feliz", "triste", "sorpresa"]
        self.detector = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_front
```

Después con la función cargar modelo creamos una instancia de la arquitectura CNN y cargamos los pesos entrenados desde el archivo .pth, Ponemos el modelo en modo evaluación desactivando dropout y retornamos el modelo al dispositivo adecuado.

```
def cargarModelo(self, rutaModelo):
    modelo = EmocionCNN(num_classes=5)
    modelo.load_state_dict(torch.load(rutaModelo, map_location=self.dispositivo))
    modelo.eval()
    return modelo.to(self.dispositivo)
```

En la función detectar emociones lo primero que hacemos es ajustar la imagen del rostro a 48x48 y mandamos a llamar las transformaciones y le pasamos la img del rostro que está capturando. Despues con la predicción desactiva cálculo de gradientes para eficiencia, también obtiene salidas del modelo y selecciona la emoción con mayor probabilidad.

```
def detectarEmocion(self, imgCara):
    imgCara = cv2.resize(imgCara, (48, 48))
    tensor_img = self.transformaciones(imgCara).unsqueeze(0).to(self.dispositivo)

    with torch.no_grad():
        salidas = self.modelo(tensor_img)
        _, predicted = torch.max(salidas, 1)
        emocion = self.emociones[predicted.item()]

    return emocion
```

Después tenemos la función video la cual captura el frame de la cámara y convierte a escala de grises y detecta rostros utilizando el clasificador de cascada, para cada cara recorta la región del rostro, clasifica la emoción y dibuja un rectángulo con un color representativo de la emoción, por ejemplo verde para feliz y sorpresa, rojo para enojado, angustia y triste y por ultimo muestra el frame procesado.

```
def interfaz(self):
    cap = cv2.VideoCapture(0)

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        gris = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        caras = self.detector.detectMultiScale(gris, 1.3, 5)

        for (x, y, w, h) in caras:
            imgCara = gris[y:y+h, x:x+w]
            emocion = self.detectarEmocion(imgCara)

            color = (0, 255, 0) if emocion in ["feliz", "sorpresa"] else (0, 0, 255)
            cv2.rectangle(frame, (x, y), (x+w, y+h), color, 2)
            cv2.putText(frame, emocion, (x, y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, color, 2)

        cv2.imshow('Detectar Emociones', frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
```

Link de pruebas video drive: https://drive.google.com/file/d/1UzuRvOq2PkI-9wVSfEoSz7PffXXgPuxh/view?usp=sharing

Resultados