Extracción de características en imágenes

Inferencia de dígitos en MNIST con HOG, LBP y SVM

Autor:
Jesús Enrique Cartas Rascón

Profesores:
Jesús Chamorro Martínez
Rafael Molina Soriano

Índice general

CAPÍTULO 1

Ejercicio base

El ejercicio base consiste en tratar de predecir los dígitos correspondientes a las imágenes del dataset MNIST [1]. Concretamente, hemos de predecir únicamente los últimos dos dígitos que aparezcan en el DNI de cada uno. En mi caso estos dígitos son 0 y 1.

Para ello, se creará un histograma de gradientes [2] de la imagen y esta información se pasará como datos de entrenamiento para una Support Vector Machine. [3]

Este primer estudio se centrará en evaluar el rendimiento de los diferentes kernels que puede tomar la SVM, al menos los disponibles en la librería openCV. Estos son `cv2.ml.SVM_LINEAR`, `cv2.ml.SVM_POLY`, `cv2.ml.SVM_RBF`, `cv2.ml.SVM_SIGMOID`

1.1. Diseño del algoritmo y rendimiento

El tiempo total de la ejecución tomó **728.58 segundos**, lo que corresponde a algo más de 12 minutos. Sin embargo, la distribución de tiempo no fue en absoluto regular, ya que hubo gran diferencia de tiempo dependiendo del kernel.

Se han acometido toda serie de optimizaciones (en la medida de lo posible dada la complejidad algorítmica y mis actuales conocimientos) para lograr rebajar el tiempo de ejecución lo máximo posible.

En las primeras versiones funcionales, la ejecución de una sola etapa de validación tomaría los 12 minutos completos, lo cual estima que, cinco veces eso, suma hasta una hora. Pero en esa ocasión, se ejecutaba un kernel para cada uno de las etapas de validación, lo que significa que los últimos kernels en ejecutar tardarían mucho más de 12 minutos.

En la versión optimizada se ejecutan todos los kernels a la vez para cada etapa de validación, entonces la duración de cada iteración es la duración del fold que estemos ejecutando.

En resumen, antes se ejecutaba un algoritmo que puede representarse como:

```
for every kernel do
    for every fold do
        classifier = model.train(X_train, y_train);
        predictions = classifier.test(X_test);
        accuracy = score(predictions, y_test);
    end
end
```

Algorithm 1: Primera versión no optimizada

Y la versión optimizada puede representarse tal que:

```
for every fold do
    classifiers = pool.map(model.train, kernels);
    predictions = pool.map(test, classifiers);
    accuracy = score(predictions, y_test);
end
```

Algorithm 2: Versión actual optimizada

Eliminar el bucle de los kernels gracias a la paralelización parcial de la ejecución de los modelos reduce considerablemente el tiempo de ejecución. Además, se observó que los modelos con diferentes kernels también se ejecutaban en menos tiempo individualmente.

# Fold	Linear	Polynomial	RBF	Sigmoid
1	**100**	99.96	**100**	83.26
2	99.88	99.84	99.92	80.41
3	**100**	99.96	93.13	88.74
4	99.96	**100**	86.14	86.30
5	**100**	**100**	88.94	87.12
Test	**99.95**	99.95	94.17	88.68

Cuadro 1.1: Tabla resumen de los resultados por etapa de validación y kernel.

1.2. Resultados

En la tabla 1.1 se observan los resultados obtenidos para cada kernel y para cada etapa de validación. Se optó por una clásica 5-fold-cv, lo que nos deja en 5 etapas.

La medida de precisión es tan simple como el ratio de acierto/total de respuestas. El valor que aparece en las etapas de validación corresponde al test que se hace con la partición correspondiente en esa etapa, que, como sabemos, sigue perteneciendo al conjunto de entrenamiento.

La última fila ya sí muestra la etapa de test real, con imágenes que el modelo no ha visto en ningún momento del entrenamiento.

Vemos que el modelo **lineal**, que fue el que menos tardó con diferencia, obtiene un resultado prácticamente inmejorable. El sigmoide, que fue el que más tardó, no está a la altura.

Hay que destacar que existe otro modelo de kernel, `cv2.ml.CHI2`. Se decidió eliminar este modelo ya que, en entrenamientos previos con conjuntos pequeños obtenía incluso peores resultados, pero el tiempo de ejecución era desorbitado. No tiene sentido utilizar ese kernel, al menos, por supuesto, para nuestra tarea.

CAPÍTULO 2

Implementación: Local Binary Patterns

En este capítulo discutiremos cómo se ha acometido la implementación del algoritmo Local Binary Patterns y de las consideraciones tomadas durante su desarrollo. Además, compararemos los resultados de nuestro modelo con respecto al uso de HOG.

2.1. Diseño e implementación

Discutiremos en esta sección las justificaciones de la última versión desarrollada del algoritmo y el por qué de las decisiones tomadas.

2.1.1. Versión original

Como ya sabemos, el algoritmo LBP toma como entrada una imagen, y devuelve el histograma de otra imagen del mismo tamaño. Esta imagen generada contiene información acerca de los píxeles que se encuentran en potenciales fronteras, y las distintas secciones del histograma nos informan de las combinaciones de bordes y esquinas presentes.

Para ello, lo clásico sería utilizar un algoritmo secuencial que recorra todos los píxeles, y en cada píxel ejecute la operación de calcular el valor binario.

Esta operación toma el vecindario del píxel y genera un número binario para cada vecino, en sentido de las agujas del reloj empezando desde la esquina superior izquierda. Comparamos cada vecino con el pixel central y, si es mayor, añadimos un 1, 0 en otro caso. Esto nos devuelve un número en binario que, convertido a decimal, ocupará la posición del pixel central en la nueva imagen.

Posteriormente, calculamos el histograma de esta imagen. Si el número de vecinos es 8, el mayor número que se puede representar con 8 dígitos binarios es 255, lo que coincide con el máximo nivel en una imagen en escala de grises. Por lo tanto, confeccionaríamos un histograma de 256 *bins* y finalmente lo devolvemos.

```
img = empty matrix;
for i in rows do
    for j in columns do
        img[i,j] = binary_neighbours(input_image[i, j]);
    end
end
return histogram(img);
```

Algorithm 3: Versión original del LBP para una imagen dada.

Este método funcionaría bien, pero sería extremadamente lento, teniendo en cuenta que, de por sí, Python no es un lenguaje que se centre en el rendimiento. Eso haría del estudio de los algoritmos una tarea muy tediosa, teniendo que esperar varias horas para cada ejecución.

2.1.2. Versión optimizada

Mi propuesta hace uso de las ventajas de la programación funcional y la paralelización, evitando al máximo los bucles `for` y aprovechando bien los recursos.

En primer lugar, tomamos todas las posibles *ventanas* que nuestra imagen tiene. Si lo pensamos, iterando como en el algoritmo 3, en cada iteración no tomamos solo el píxel, sino su vecindario. Recorriendo todos los píxeles, habremos recorrido efectivamente toda la imagen en las *ventanas* que cada pixel necesita para su cálculo.

Por ello, podemos obtener todas esas posibles ventanas directamente con la función `sliding_window_view`. Esta función recibe la imagen y una tupla que corresponde a las dimensiones de las ventanas que queremos tomar. Nos devuelve **una lista con todas las posibles ventanas** que se recorrerían en caso de iterar como en el algoritmo original.

Esto es importante, porque teniendo una lista y una función, podemos hacer uso de la técnica `list comprehension` en python. Está rigurosamente estudiado que esta técnica funciona mucho más rápido para iterar que un ciclo `for`, al menos para números grandes de iteraciones.

Así, podemos obtener el valor correspondiente de cada ventana simplemente llamando a la función `binary_neighbours`.

windows = **sliding_window_view**(input_image, (3, 3));
lbp_image = [**binary_neighbours**(window) for window in windows];
return histogram(lbp_image);

Algorithm 4: Versión optimizada del LBP para una imagen dada.

Es importante tener estas consideraciones ya que es una operación que se va a ejecutar cientos de veces por imagen. Concretamente en nuestro caso, 784 (28×28). Eso luego se repite para miles de imágenes, algo más de 12000 en nuestro caso. Vemos que, muy rápidamente, el número escala a valores desorbitados.

De no tomar estas decisiones, el cálculo de los histogramas podría efectivamente durar horas, y eso no es ideal.

2.2. Resultados

Ahora que sabemos cómo funciona el algoritmo, veamos sus resultados.

Realmente, solo estamos sustituyendo el algoritmo de HOG por el LBP para calcular los gradientes. La máquina SVM funciona de forma exactamente igual en ambos estudios. Se pretende analizar cómo de bien lo hace una determinada configuración de la SVM ante distintas técnicas de cálculo de gradiente, para ver cuánta información provee cada uno.

Se puede observar en la tabla 2.1 que los resultados son, cuanto menos, extraños.

El modelo Lineal lo hace perfectamente en todas las iteraciones excepto en la última. El modelo Polinomial cae en la segunda iteración. RBF y Sigmoid siguen sin estar a la altura, pero han obtenido resultados poco consistentes.

El modelo Sigmoid ha obtenido un resultado de 0 en las primeras dos fases, lo cual es, estadísticamente hablando, igual de difícil que obtener un 100 %. Pero posteriormente ha obtenido un 100 %. Si hubiera obtenido un 0 % consistente, al menos podríamos obtener un modelo de alta precisión simplemente negando la predicción que nos diera. Pero al obtener un buen resultado después, se rompe esta posibilidad.

Es destacable que, en la fase de test, tampoco hay consistencia. El modelo Lineal ha caído, y RBF y Sigmoid han respondido al azar, obteniendo un 50 %. Estadísticamente hablando, en un ejercicio como este en el que la respuesta es binaria (0 o 1, literalmente), obtener un 50 % es, en realidad, responder aleatoriamente.

# **Fold**	**Linear**	**Polynomial**	**RBF**	**Sigmoid**
1	98.46	98.57	0.03	0
2	96.48	7.26	0.03	0
3	99.13	99.32	66.20	66.16
4	99.13	96.21	**100**	**100**
5	32.49	96.92	**100**	**100**
Test	37.0	**98.0**	50.0	50.0

Cuadro 2.1: Tabla resumen de los resultados por etapa de validación y kernel.

En cuestiones de rendimiento, para el mismo exacto ejercicio (12k imágenes de train y 2.1k de test) y 5 etapas de validación, esta versión con LBP tarda, sorprendentemente, algo menos de cuatro minutos.

En comparación con los 12 del HOG, es un tercio del tiempo, pero la distribución de los tiempos no es igual.

En HOG, el tiempo que se toma en procesar todas las imágenes (es decir, obtener sus histogramas) es básicamente insignificante. Se nota que la librería de openCV está bien optimizada para aplicaciones en tiempo real. Sin embargo, la SVM tarda mucho más en aprender las características de los histogramas de gradientes.

En el caso de LBP, la mayor parte del tiempo de ejecución del programa es, esencialmente, cálculo de histogramas. Como digo, esto podría ser mucho peor en caso de no tener estas consideraciones. La SVM es capaz de procesar los histogramas del LBP mucho

mejor ya que, la dimensión es mucho menor tambien: 256 números en LBP vs algo más de 4k en HOG.

2.3. Conclusiones

Podemos concluir que el método de histogramas de gradiente orientados, si bien toma proporcionalmente más tiempo, también ofrece resultados muy consistentes.

El método LBP no ha ofrecido malos resultados, pero no podemos confiar debido a la tan alta varianza que ofrece entre etapas y en test.

CAPÍTULO 3

Manual de uso

Para poder usar el programa desarrollado, primero hemos de descargarlo.

Utilizando la terminal, podemos ejecutar

```
git clone https://github.com/jesi-rgb/extraccion-rasgos
```

para clonar el repositorio en el directorio en el que nos encontremos.

También debemos descargar la carpeta `mnist_data`, que, no se ha incluido en el repositorio por razones de almacenamiento. La carpeta `mnist_data` deberá alojarse en el root del directorio del proyecto, a la altura de la carpeta `src`.

Una vez configurado, tendremos un directorio llamado `extraccion-rasgos` del que cuelgan las carpetas `src` y `mnist_data`, así como algunos archivos.

Haciendo uso de los entornos virtuales de python, podemos crear uno como mejor nos convenga (conda o python) y rápidamente instalar las dependencias, con el comando

```
pip install -r requirements.txt
```

Hecho todo esto, estamos listos para poder ejecutar el programa.

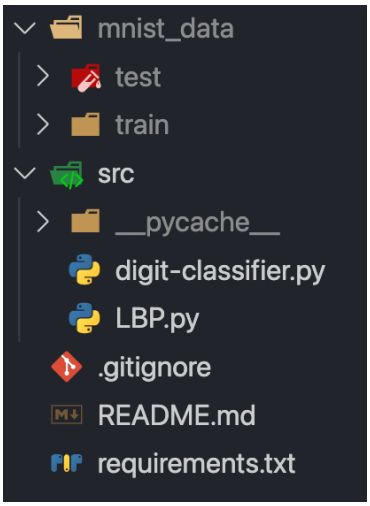

Figura 3.1: Ejemplo de la estructura del repositorio completa.

Situandonos en el directorio root, es decir, `extraccion-rasgos`, podemos ejecutar

```
python src/digits-classifier.py
```

para comenzar la ejecución por defecto.

Es importante ejecutarlo así (desde el directorio raíz y accediendo a src mediante el comando), de otra manera las rutas internas no funcionarán.

3.1. Parámetros

Tenemos 4 parámetros disponibles para personalizar la ejecución:

- `-hist`. Valores: hog, lbp.

  Este argumento nos permite utilizar el método HOG o el LBP.

  Por defecto: hog.

- `-f`. Valores: int.

  Determina el número de etapas de validación usadas en el cross-validation.

  Por defecto: 5.

- `-c`. Valores: int en rango 1-10.

  Determina cuántas clases vamos a tomar del conjunto total. Toma las c primeras clases en orden. Esto es, si lo ejecutamos con c = 3, tomaremos las clases zero, one and two.

Por defecto: 2 (zero and one).

- `-k`. Valores: int.

  Determina cuántos samples vamos a tomar para el conjunto de entrenamiento y test para cada clase. Es decir, si pasamos 500, tomaremos 500 fotografías de la carpeta zero, 500 de la one, y así, hasta el número de clases que hayamos especificado en -c.

  Por defecto: -1 (tomar todas las imágenes disponibles)

Un ejemplo de uso sería:

```
python src/digits-classifier.py -hist lbp -c 4 -k 700
```

que indica que usaremos el algoritmo LBP, tomaremos las cuatro primeras clases (zero, one, two and three) y que tomaremos 700 samples de cada clase para entrenar y testear.

Bibliografía

[1] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[2] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, 1:886–893, 2005.

[3] Marti A. Hearst. Support vector machines. *IEEE Intelligent Systems*, 13(4):18–28, July 1998.