PRÁCTICA 2.
REDES NEURONALES.

Luis Nolasco Ramírez
Grado de ingeniería Robótica UA

Preparación del dataset

Toda red neuronal necesita ser entrenada para poder cumplir sus especificaciones, para ello es necesario preparar los datos de entrada de dicha red creando un archivo que se usará como “dataset”. Este archivo contiene los datos de todas las variables de entrada y de su clasificación en la red.

Para este proyecto las variables de entrada son imágenes, para crear el dataset se usará una matriz que tendrá tantas filas como variables de entrada, es decir, imágenes ; y tantas columnas como píxeles más tres columnas de clasificación que contienen valores de 0 o 1 en función de su pertenencia a dicha clase.
Las últimas tres columnas contienen un máximo de tres valores distintos, uno por cada clase correspondiente: “1 0 0” para la clase “Bird”, “0 1 0” para la clase “Dog” y “0 0 1” para la clase “Horse”.
Cada imagen se almacena en una sola fila de dicha matriz, para ello es necesario convertir cada imagen en un vector de una sola fila.
Se van a obtener dos dataset distintos, uno del canal V del espacio de color HSV de la imagen, la cual es la imagen en blanco y negro, y otro del canal H.

Las imágenes utilizadas para la creación de dicho dataset se encuentran en la carpeta de nombre dataset. Cuando se selecciona desde la interfaz la opción “Generate Dataset” se abre una ventana para seleccionar la carpeta donde se encuentran las imágenes, una vez seleccionada se crea una lista con los nombres de todos los archivos contenidos en esa carpeta haciendo uso de la función Qt “entry.List()”. Con un bucle se recorre la totalidad de la lista de archivos rellenando la matriz.
En cada iteración se recorre la imagen al completo y para cada píxel de la imagen se insertan como un nuevo elemento al final de sus correspondientes vectores “img_vectorV” y “img_vectorH”. Una vez añadidos todos los píxeles se añaden las tres últimas columnas con su correspondientes clase que se obtiene leyendo el nombre del archivo y comprobando cuál de los tres posibles nombres contiene: “Bird”,”Dog”,”Horse; usando el método “contains” de la clase QstringList.

Una vez creada la matriz completa se almacena en dos archivos “my_datasetH.csv” y “my_datasetV.csv” haciendo uso de las clases Qfile y QtextStream. Con ellas se crean ambos archivos y se abren como streams de salida los cuáles se van rellenando con los valores y se separan con comas.

OpenNN utiliza también el formato xml para la clasificación de datasets, por ello es necesario crear un archivo xml para cada csv. Con la clase DataSet de OpenNN se abre el archivo csv, con el método “get_variables_pointer()” se obtiene el puntero a las variables del DataSet. Inicialmente, la última columna es interpretada como salida y el resto de variables como entrada, es por ello que se debe cambiar el nombre y el tipo de variable de las variables de salida que no están definidas. Conociendo su índice en el vector de cada imagen es muy sencillo usando los métodos “set_name()” y “set_use()”, los índices de las clases se encuentran en las últimas tres columnas, la primera de las tres se encuentra en el índice que ocupa la posición de número igual al total de píxeles de la imagen.
Solo queda dividir el conjunto de datos del DataSet para entrenamiento, validación y test. Esto se define en el punto de instancias que almacena los índices a usar en cada conjuntos. Se pueden repartir esos índices de forma aleatoria con el método “split_random_indices()” indicando en él cual es el porcentaje total del DataSet a usar en cada conjunto.
De esta forma ya se tienen todos los datos de entrada en el archivo csv y la información y distribución del DataSet en el archivo xml.

Implementación de las redes neuronales

El primer paso es definir la estructura de la red, es decir, sus capas. En este caso se define un perceptrón multicapa, una red neuronales formada por múltiples capas, con esto se consigue representar funciones no lineales, este perceptrón incluye una capa de entrada, una capa de salida y una o más capas ocultas. Para ello se define un vector de enteros de nombre “layers”, cada elemento de dicho vector indica el número de neuronas en dicha capa, siendo el primer elemento la capa de entrada que tendrá tantas neuronas como elementos de entrada (ancho_de_imagen*alto_de_imagen en este caso) y el último elemento la capa de salida que tiene tantas neuronas como posibles salidas o clases (3 en este caso). Con este vector y la clase MultilayerPerceptron se define el perceptrón y con dicho perceptrón y la clase NeuralNetwork se define la red neuronal.
Para problemas de clasifiación la red suele estar formada por una capa de escalado, 1 o 2 capas ocultas y una capa probabilística, además de la entrada y la salida. Más adelante se explicará más en profundidad la implementación de dichas capas.

Una vez se ha definido la estructura de la red y, por ende, la propia red, es necesario indicar la información de entrada y la información de salida. Esta información está definida en el data_set.csv. Se carga el DataSet y se obtiene el puntero a las variables con “get_variables_pointer()”. Con este puntero se tiene pleno acceso a todas las direcciones de memoria de las variables.
Usando el método “arrange_inputs_information()” en la variable del DataSet se obtienen las entradas y se almacenan en una matriz. Ahora se obtiene el puntero a las entradas de la red con “get_imputs_pointer()” y con dicho puntero y el método “set_information()” se asignan a la red las entradas obtenidas desde el DataSet.
De forma análoga para las salidas, con el método “arrange_targets_information()” en la variable del DataSet se obtienen las salidas y se almacenan. Con “get_outputs_pointer()” en la red y “set_information()” se asignan las salidas del DataSet a la red.

Capa de escalado

Escalar los datos permite a la red neuronal trabajar en mejores condiciones. El método “scale_inputs_minimum_maximum()” de la case DataSet escala el DataSet entre un valor mínimo y un máximo y devuelve una estructura que contiene la información de entrada y el escalado. Esta estructura contiene valores como el máximo, mínimo, media y desviación estándar y será mencionada a partir de ahora como vector de estadística.
Una vez definido este vector de estadística se crea la capa de escalado con el método “construct_scaling_layer()” de la clase NeuralNetwork, con “get_scaling_layer_pointer()” se

obtiene el puntero a la dirección de memoria que almacena las características de dicha capa. Las más importantes y que se deben definir son: la estadística y el método de escalado.
La estadística está definida en el vector de estadística y se asigna con el método “set_statistics()”. Debido a que el DataSet ya ha sido escalado con anterioridad se define el método de escalado de esta capa como “NoScaling”

Capa probabilística
Coge las salidas y produce nuevas salidas cuyos elementos se interpretan como probabilidades. Los resultados están en el rango de 0 a 1 y la suma de todos ellos es siempre igual a 1. Se puede modificar la función de activación de las neuronas de esta capa con un método implementado para la propia capa.
Esta capa se crea con el método “construct_probabilistic_layer()” de la clase NeuralNetwork. Con “get_probabilistic_layer_pointer()” se obtiene el puntero a la capa y con “set_probabilistic_method()” se especifica la función de activación de las neuronas de la capa.

En ambos casos ninguna de estas capas es obligatoria pero es muy recomendable su uso en problemas de clasificación como en el que se presenta en esta práctica. Usando como base la siguiente estructura: capa de escalado, n capas ocultas, capa probabilística y capa de salida; se desarrollarán y probarán diversas redes neuronales.

Una vez se ha definida la estructura de la red es necesario llevar a cabo su entrenamiento, el entrenamiento de una red neuronal está definido en OpenNN por dos clases abstractas: LossIndex (tipo de error) y OptimizationAlgorithm (algoritmo de entrenamiento). La correcta selección de estos dos métodos es un paso clave para el entrenamiento de una red neuronal.

LossIndex
Se crea un objeto de la clase LossIndex y se definen como entradas la red neuronal y el data set, con el método “set_error_type()” de dicha clase se define el tipo de error a utilizar. Entre ellos podemos encontrar: SUM_SQUARED_ERROR, MEAN_SQUARED_ERROR, NORMALIZED_SQUARED_ERROR, MINKOWSKI_ERROR, WEIGHTED_SQUARED_ERROR, CROSS_ENTROPY_ERROR.

TrainingStrategy
Se crea un objeto de la clase TrainingStrategy que usa como entrada el objeto de la clase LossIndex, usando el método “set_main_type()” se define el algoritmo de entrenamiento. Entre ellos: GRADIENT_DESCENT, CONJUGATE_GRADIENT, QUASI_NEWTON_METHOD, LEVENBERG_MARQUARDT_ALGORITHM, STOCHASTIC_GRADIENT_DESCENT, ADAPTIVE_MOMENT_ESTIMATION.

Ahora solo queda definir algunos parámetros respecto al entrenamiento, se obtiene la variable puntero del algoritmo de entrenamiento seleccionado, el método varía en función del algoritmo elegido, y se establecen los parámetros deseados. Existen más parámetros pero para este caso se va a configurar el número máximo de iteraciones, el número de periodos tras los que se van mostrando los avances del entrenamiento y el tiempo máximo de entrenamiento, con los métodos “set_maximum_iterations_numer()”, “set_display_period()” y “set_maximum_time()” respectivamente. Seguidamente se inicia el entrenamiento con “perform_training()” de la clase TrainingStrategy.

Finalizado el entrenamiento se guarda la red neuronal en un fichero xml. Ahora solo quedar comprobar su precisión, OpenNN tiene implementada la clase TestingAnalisis para este objetivo. Se crea un objeto de la clase TestingAnalisis cuyos parámetros de entrada son la red neuronal y el

dataset, con el método “calculate_confusion()” de dicha clase se calcula la matriz de confusión de la red. Teniendo dicha matriz, la precisión de la red se puede calcular como la traza de la matriz entre la suma de sus elementos. Tanto la traza como la suma se pueden calcular de forma sencilla con métodos de la clase Matrix.

Análisis y comparación de los resultados de entrenamiento y validación

Las redes implementadas y sus correspondientes resultados son las siguientes:
*Cada red se ha probado para los dos datasets obtenidos, el del canal V y del canal H del espacio HSV. A su vez, para cada dataset se ha probado con dos distribuciones distintas de los datos de entrenamiento.

Red 1. Estrategia de búsqueda Quasi_Newton, tipo de error Mean_Squared, método probabilístico softmax, método de escalado noScaling. 4 capas de 1024, 10, 3 y 3 neuronas respectivamente. 100 épocas de entrenamiento.

RED 1		
DATASET	DISTRIBUCIÓN	PRECISIÓN
DatasetV	0.6, 0.2, 0.2	39.62%
DatasetH	0.6, 0.2, 0.2	35.84%
DatasetV	0.8, 0.1, 0.1	35.84%
DatasetH	0.8, 0.1, 0.1	40.7407%

Red 2. Estrategia de búsqueda Quasi_Newton, tipo de error Mean_Squared, método probabilístico softmax, método de escalado noScaling. 4 capas de 1024, 15, 3 y 3 neuronas respectivamente. 100 épocas de entrenamiento.

RED 2		
DATASET	DISTRIBUCIÓN	PRECISIÓN
DatasetV	0.6, 0.2, 0.2	37.7359%
DatasetH	0.6, 0.2, 0.2	32.8755%

Red 3. Estrategia de búsqueda Gradient Descent, tipo de error Mean_Squared, método probabilístico softmax, método de escalado noScaling. 4 capas de 1024, 10, 3 y 3 neuronas respectivamente. 125 épocas de entrenamiento.

RED 3		
DATASET	DISTRIBUCIÓN	PRECISIÓN
DatasetV	0.6, 0.2, 0.2	45.283%
DatasetH	0.6, 0.2, 0.2	33.9623%
DatasetV	0.7, 0.2, 0.1	29.5707%
DatasetH	0.7, 0.2, 0.1	33.3333%

Red 4. Estrategia de búsqueda Quasi_Newton, tipo de error Weighted_Squared, método probabilístico softmax, método de escalado noScaling. 4 capas de 1024, 10, 3 y 3 neuronas respectivamente. 100 épocas de entrenamiento.

RED 4		
DATASET	DISTRIBUCIÓN	PRECISIÓN
DatasetV	0.6, 0.2, 0.2	45.283%
DatasetH	0.6, 0.2, 0.2	35.8491%
DatasetV	0.7, 0.2, 0.1	44.4444%
DatasetH	0.7, 0.2, 0.1	37.037%

Red 5. Estrategia de búsqueda Quasi_Newton, tipo de error sum_squared, método probabilístico softmax, método de escalado noScaling. 4 capas de 1024, 10, 3 y 3 neuronas respectivamente. 100 épocas de entrenamiento.

RED 5		
DATASET	DISTRIBUCIÓN	PRECISIÓN
DatasetV	0.6, 0.2, 0.2	32.0755%
DatasetH	0.6, 0.2, 0.2	32.0755%

No se ha seguido probando esta red debido a los pésimos resultados obtenidos, se puede concluir que el error sum_squared no es el adecuado para este proyecto.

Red 6. Estrategia de búsqueda Gradient Descent, tipo de error Mean_Squared, método probabilístico softmax, método de escalado noScaling. 6 capas de 1024, 10, 10, 10, 3 y 3 neuronas respectivamente. 150 épocas de entrenamiento. Se ha modificado la función de activación de las primeras capas de forma que la capa 0 y la capa 2 tienen una función de activación del tipo “Logistic” y la capa 1 una función del tipo “HyperbolicTangent”. 150 épocas de entrenamiento.

RED 6		
DATASET	DISTRIBUCIÓN	PRECISIÓN
DatasetV	0.6, 0.2, 0.2	45.283%
DatasetH	0.6, 0.2, 0.2	45.283%
DatasetV	0.7, 0.2, 0.1	44.4444%
DatasetH	0.7, 0.2, 0.1	25.9259%

Es importante tener en cuenta el tiempo que toma cada algoritmo para entrenar la red, mientras que el algoritmo Quasi Newton toma casi 1 hora para entrenar las redes 1 y 2 de forma individual, al algoritmo Gradient Descent le toma apenas 5 minutos entrenar la red 3. Debido a este aspecto se probó a aumentar considerablemente el número de épocas que empleaba el algoritmo Gradient Descent, por desgracia para valores superiores a 125 se produce sobreentrenamiento y los resultados empeoran cada vez más.
También se puede apreciar que, excepto casos contados, las redes producen mejores resultado cuando toman como entrada el datasetV en vez del datasetH, se puede concluir entonces que con el

canal V del espacio HSV (el cual se puede interpretar como una representación en blanco y negro de la imagen) queda mejor representada la imagen.

A continuación se van a comparar los resultados más interesantes obtenidos en las distintas redes:

Red	Dataset	Distribución	Algoritmo	Error	Capas	Extras	Precisión%
1	V	0.6, 0.2, 0.2	Quasi Newton	Mean Squared	1024, 10, 3,3		39.62
1	V	0.8, 0.1, 0.1	Quasi Newton	Mean Squared	1024, 10, 3,3		40.7407
3	V	0.6, 0.2, 0.2	Gradient Descent	Mean Squared	1024, 10, 3,3		45.283
4	V	0.6, 0.2, 0.2	Quasi Newton	Weighted Squared	1024, 10, 3,3		45.283
4	V	0.7, 0.2, 0.1	Quasi Newton	Weigthed Squared	1024, 10, 3,3		44.4444
6	V	0.6, 0.2, 0.2	Gradient Descent	Mean Squared	1024, 10, 10, 10, 3, 3	Cambio en la función de activación	45.283
6	H	0.6, 0.2, 0.2	Gradient Descent	Mean Squared	1024, 10, 10, 10, 3, 3	Cambio en la función de activación	45.283

En el caso de la red uno se aprecia una ligera mejora al cambiar la distribución del dataset aunque nada significativo, si que se observa un cambio más relevante cuando en redes que utilizan Quasi Newton se cambia el tipo de error a Weighted Squared, se observa una mejora. Gradient Descent obtiene resultados aceptables con la definición más básica de la red, pero todas las modificaciones realizadas para mejorar su funcionamiento derivan en peores resultados. Se ha probado a aumentar el número de neuronas por capa, cambiar el error o cambiar la distribución de los datos pero sin ninguna mejora. Excepto en un caso, aunque no se ha obtenido mejora con el datasetV se ha conseguido que la precisión usando el datasetH sea la misma que con el otro dataset, este es el caso de la red 6, añadiendo dos capas más con 10 neuronas cada una y modificando la función de activación de las primeras capas se consigue mejorar la precisión de la red para el datasetH. Aunque no se haya mejorado la precisión del datasetV la mejora de la precisión del datasetH es una buena noticia pues supone mayor generalización de la red.
En base a todas las pruebas, la red que ha quedado implementada en el código es la número 6, además de por tener la mayor precisión para ambos dataset, también por el hecho de que el tiempo de entrenamiento requerido por Gradient Descent es mucho menor. Para esta cantidad de datos es irrelevante, pero es un punto a tener en cuenta y que decanta la balanza a su favor.

Cada red neuronal de las mencionadas tiene como nombre “final_neural_network_x_y.xml” siendo “x” el número que corresponde a la red a la que pertenece e “y” otro número que representa el orden en la lista. Ejemplo: “final_neural_network_6_1.xml” contiene el segundo elemento de la lista de redes de la red 6.