UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Evaluación de prestaciones de las redes de cápsulas matriciales sobre distintos escenarios de clasificación de imagen

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA EN SISTEMAS DE TELECOMUNICACIÓN

CURSO: 2018/2019

Autor: JUAN ANTONIO GARCÍA CAMPILLO

Director: JORGE LARREY RUIZ

”*Si es delito no saber, señoría lléveme preso.”*

Arnau Griso

Agradecimientos

A mis padres y mi hermana y mi cuñado por darme siempre su apoyo y por creer en mi más que yo mismo.

A mis compañeros Antonio Oliva y Martín Prieto por ofrecerme su ayuda siempre que se la he pedido.

A mi amigo Iván, por ofrecerme su ordenador cuando el mío no era lo suficientemente potente, por ayudarme en todo lo que ha podido siempre y por ser mi apoyo en las vacas flacas.

A Mayka, por acompañarme en tantos momentos en este largo camino de la carrera y porque me ha hecho sentir mejor persona de lo que soy.

Al director de este proyecto Jorge Larrey por brindarme la oportunidad de realizar este trabajo y por darme siempre los medios que ha tenido en su mano para ayudarme.

Resumen

En este documento se redacta un estudio realizado sobre redes neuronales de cápsulas. Este tipo de redes fueron inventadas por un grupo de investigadores de la universidad de Toronto con Geoffrey E. Hinton a la cabeza y con la colaboración especial de Sara Sabour y Nicholas Frosst. Su invención en 2017 supuso una mejora importante en cuanto al avance de la investigación del *machine learning*. Cabe destacar que las redes de cápsulas se pueden dividir en dos grupos: las redes de cápsulas vectoriales y las matriciales, siendo las cápsulas matriciales una evolución de las otras y publicadas en 2018. Los siguientes apartados trataran en primer lugar, unos conceptos básicos de aprendizaje máquina y de las limitaciones de las redes neuronales anteriores, por las cuales la comunidad científica comenzó a investigar en el avance de nuevas tecnologías, hasta dar lugar a las redes de cápsulas. Posteriormente, se explicarán teóricamente los tipos de redes de cápsulas. Para finalizar, se realizará una comparación de estas redes de manera experimental.

Abstract

In this document a study which has been carried out on capsule neural networks is written. These types of networks were invented by a group of researchers from Toronto University with Geoffrey E.Hinton at the head of it and with special collaboration from Sara Sabour y Nicholas Frosst. Their invention in 2017 was an important improvement regarding the advances in the research of the machine learning. It should be noted that capsule neural networks can be divided into two groups: vector capsules and matrix capsules, being matrix capsules an evolution of the other ones and published in 2018. The following sections are first of all about basic concepts of machine learning and limitations of previous neural networks, whereby the scientific community started to research in the advance in new technologies to result in capsule networks. Subsequently, the types of capsule networks will be explained theoretically. Finally, a comparison of these networks will be made experimentally.

Tabla de contenido

Tabla de figuras

1. Introducción: Aprendizaje máquina (machine learning)

El **aprendizaje máquina** es una rama de inteligencia artificial, cuyo objetivo primordial es desarrollar algoritmos o técnicas que permiten que una computadora aprenda, a partir de un conjunto de datos de entrada, a identificar patrones y a tomar decisiones con una mínima intervención humana. Esta idea nació de la teoría que expone que las computadoras pueden aprender sin ser programadas explícitamente para realizar una tarea específica. Esto es posible debido al modelo iterativo del *machine learning*, el cual se adapta de forma independiente a medida que los modelos son expuestos a nuevos datos. Por tanto, siguiendo este modelo, una maquina aprende de cálculos previos para realizar decisiones y resultado confiables y repetibles.

Debido al desarrollo del aprendizaje máquina, se crearon las **redes neuronales** que son unas máquinas diseñadas para imitar la forma en que el cerebro realiza ciertas tareas. El cerebro humano actúa de manera muy diferente a la forma en la que lo hace un computador digital convencional. El cerebro es un sistema altamente complejo, lineal y paralelo. Este es capaz de realizar dichas tareas mediante unas celdas de computación más sencillas que realizan diversas subtareas. A estas celdas se les llama **neuronas**. Estudiando dicha neurona podemos dividirla a su vez en tres elementos más sencillos:

1. Unas **entradas ponderadas** encargadas de realizar conexiones sinápticas. La ponderación (o pesos) equivale a la fuerza o efectividad de la sinapsis.
2. Una **función de propagación** que suele ser un simple operando suma encargado de sumar todas las entradas ponderadas.
3. Una **función de activación** que se encarga de calcular el nivel o estado de activación de una neurona en función de la entrada total. Esta probablemente sea la característica más importante de una neurona dado que es la que mejor define su comportamiento. Generalmente delimitan la amplitud de la salida de una neurona. Esta función puede ser de diversos tipos, desde funciones de umbral simples, como una función signo o una función de Heaviside, a funciones no lineales, generalmente funciones sigmoidales como una función logística o una tangente hiperbólica. En el caso de redes neuronales profundas lo más habitual es utilizar la unidad **lineal rectificada (ReLU)** como función de activación. Esta función se explica con más detalle en el apartado 7.

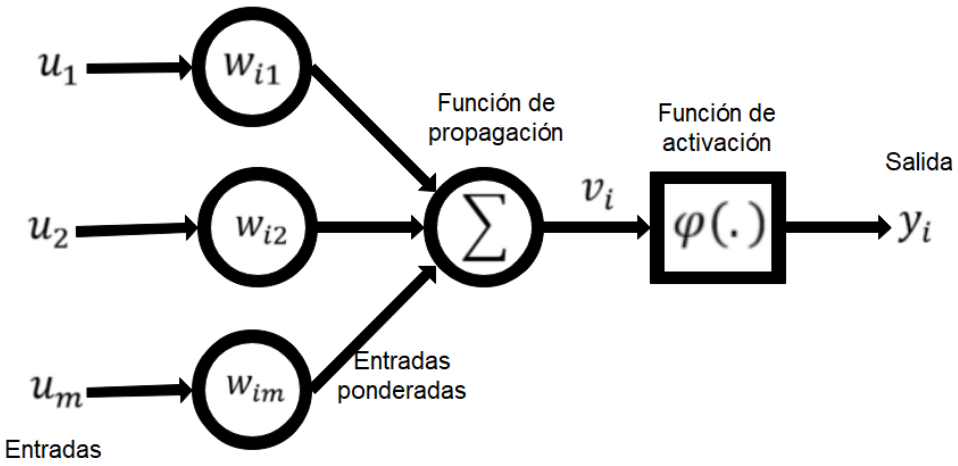

Figura 1. Estructura de una neurona.

1.1. Tipos de aprendizaje maquina

Una red neuronal se puede clasificar principalmente en una de las siguientes categorías en función de la manera que tiene de aprender, aunque puede clasificarse siguiendo muchos otros criterios:

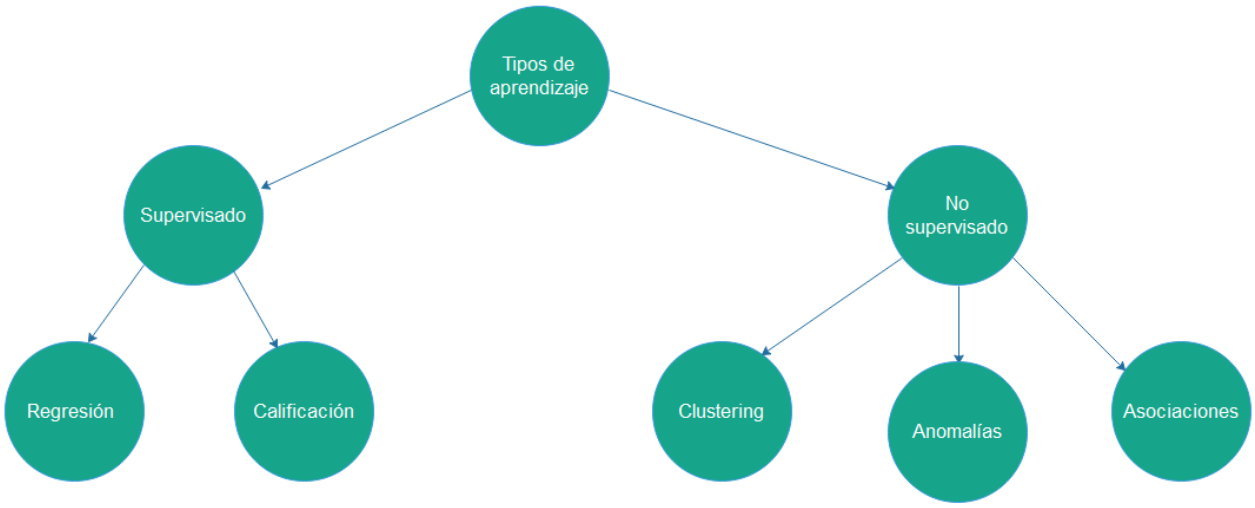

Figura 2. Tipos de aprendizaje maquina según la manera de aprender.

1.1.1. Aprendizaje supervisado

En estos métodos se parte del conocimiento a priori. A estas redes se les muestra una cantidad de datos de prueba (**ejemplos**) del tipo de conocimiento con el que se le pretende enseñar y se le muestra junto a cada ejemplo la respuesta deseada (**etiqueta**). Esta red debe de realizar lo mejor posible el mapeo entre unas entradas X y una salida Y. Cuanta más cantidad de ejemplos se le den al sistema, mejor será la precisión de este. Una vez procesados todos los ejemplos, el sistema está preparado para responder de forma autónoma, sin necesidad de adjuntar etiquetas, ante las entradas que le proporcionemos.

La variable de salida Y puede ser una variable **cuantitativa** (en ese caso estaríamos ante un problema de **regresión**) o una variable **cualitativa** (entonces, estaríamos ante un problema de **clasificación**). En los problemas de regresión se busca predecir qué valor tendrá el campo objetivo para una nueva entrada. En los problemas de clasificación, sin embargo, se pretende predecir a que categoría pertenece el campo objetivo de cada instancia a partir de una lista de posibles categorías.

Cabe destacar que, para que la red aprenda, se define una señal de error entre la respuesta deseada y la salida obtenida. Esta señal habrá que minimizarla para aumentar la precisión de nuestra red.

1.1.2. Aprendizaje no supervisado

En estos métodos, no es necesario mostrarle al sistema en ningún momento de qué trata cada ejemplo, y es el mismo sistema el que debe de desarrollar conceptos abstractos y asociarlos con cada uno de los elementos de salida. En este caso, se le proporciona a dicho sistema solamente un conjunto de datos de prueba y no se le da ninguna información de las categorías de esos ejemplos. Por tanto, este tiene que ser capaz de desarrollar **patrones** para poder etiquetar las nuevas entradas.

El aprendizaje no supervisado suele ser de tipo competitivo, puesto que solo se activa la neurona cuya respuesta ante una determinada entrada es mayor que la respuesta del resto de neuronas. La red neuronal permite encontrar patrones y regularidades estadísticas de los datos de entrada, pudiendo así extraer información relevante de estos datos y crear nuevas clases. Por tanto, se dice que este tipo de redes neuronales aprenden de forma **auto-organizada**.

Dependiendo de cuál es el objetivo del sistema podemos clasificarlo en tres tipos diferentes. El primer tipo es **agrupación o *clustering*** el cual busca entradas que son similares entre sí. Una vez creado el modelo, el sistema nos permitirá obtener a que grupo pertenece la siguiente entrada. El segundo tipo es para la **detección de anomalías**. Este es la antítesis del modelo de agrupación, ya que busca detectar entradas que se diferencien del resto. El tercer tipo son las **asociaciones**. Este tipo tiene como objetivo encontrar relaciones entre los diferentes valores que toman los campos de las entradas con el fin de deducir reglas de asociación, es decir, deducir a partir del valor determinado que toma un campo, con qué frecuencia tomará otro valor concreto.

2. Redes neuronales convolucionales

Con el paso del tiempo ha ido avanzando la tecnología y con ella han sido muchos los interesados en introducirse dentro de las Redes Neuronales Artificiales para desarrollar multitud de proyectos. Durante los últimos años, prácticamente desde que se inventaron a finales de los 80 y principios de los 90, las **Redes Neuronales Convolucionales o *Convolutional Neuronal Network*" (CNN)** han sido las punteras en este terreno. En este apartado me dispongo a hablar brevemente de algunos conceptos relacionados con estas redes, dado que nos serán útiles para entender las redes de cápsulas.

2.1. Introducción a CNN

Las CNN son redes neuronales que utilizan aprendizaje supervisado para imitar al córtex visual del ojo humano con el fin de identificar características contenidas dentro del conjunto de datos de entrada. Para ello, son necesarias varias capas ocultas especializadas y ordenadas de forma jerárquica. Es decir, las primeras capas detectan elementos simples como líneas rectas o curvas y a partir de ahí se van especializando hasta llegar a capas más profundas capaces de reconocer formas complejas como siluetas u objetos.

2.2. Funcionamiento y arquitectura de las CNN

El principio de funcionamiento de este tipo de redes se basa en el uso de la convolución de imágenes. Para entender esto mejor, lo explicaré mediante un ejemplo. Supongamos que introducimos imágenes como entrada a la red. Cada imagen tendrá de tamaño 28x28 pixeles, lo que hace un total de 784 pixeles, y cada uno de esos pixeles será asignado a un nodo de entrada. Todo esto es así siempre y cuando la imagen este en escala de grises. En el caso de que la imagen esté en color necesitaríamos 3 canales RGB (*red*, *green*, *blue*) lo que hace un total de 28x28x3=2352 nodos de entrada. Por simplicidad, tomaremos las imágenes de un solo color para nuestra capa de entrada.

La etapa que viene a continuación es una capa de **preprocesamiento**. En esta etapa se pretende normalizar los valores que pueden tomar cada uno de los pixeles. El rango inicial está comprendido entre 0 y 255. Podemos normalizar este rango dividiéndolo entre 255 para obtener valores comprendidos entre 0 y 1.

En la siguiente imagen podemos ver este preprocesamiento:

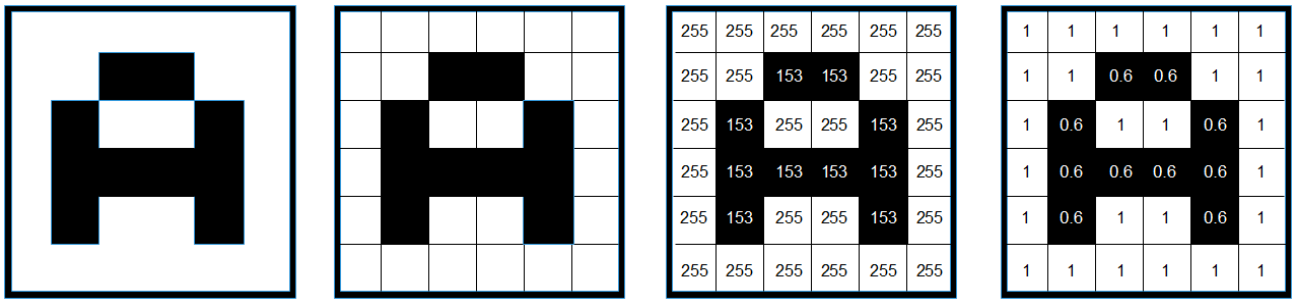

Figura 3. Preprocesamiento de una imagen.

Seguidamente, realizaremos el procesado característico de las CNN. Este se basa, como he mencionado antes, en el proceso de la **convolución.** Está consiste en tomar grupos de pixeles cercanos de la imagen de entrada e ir multiplicando escalarmente con una pequeña matriz también llamada mascara o ***kernel.*** Este *kernel* puede tener un tamaño, por ejemplo, de 3x3 pixeles, y se va moviendo por todas los nodos de entrada hasta generar una nueva matriz de salida, la cual será nuestra nueva

capa de neuronas ocultas. Cabe destacar que no usaremos tan solo un *kernel*, si no que usaremos un conjunto de ellos. A cada uno de estos *kernels* se le llama **filtro**. Por tanto, para cada uno de esos filtros, obtendríamos una matriz de salida. Por ejemplo, colocando 32 filtros, obtendríamos 32 matrices de salida. Al conjunto de todas las matrices de salida se les llama ***feature mapping*** o mapa de características. Los valores introducidos dentro de cada pixel del filtro *kernel* actuarán como los **pesos** en este proceso, ponderando los valores de los pixeles. Estos valores se ajustarán mediante el proceso de ***backpropagation*** para optimizar así el sistema. Este proceso se explicará más adelante.

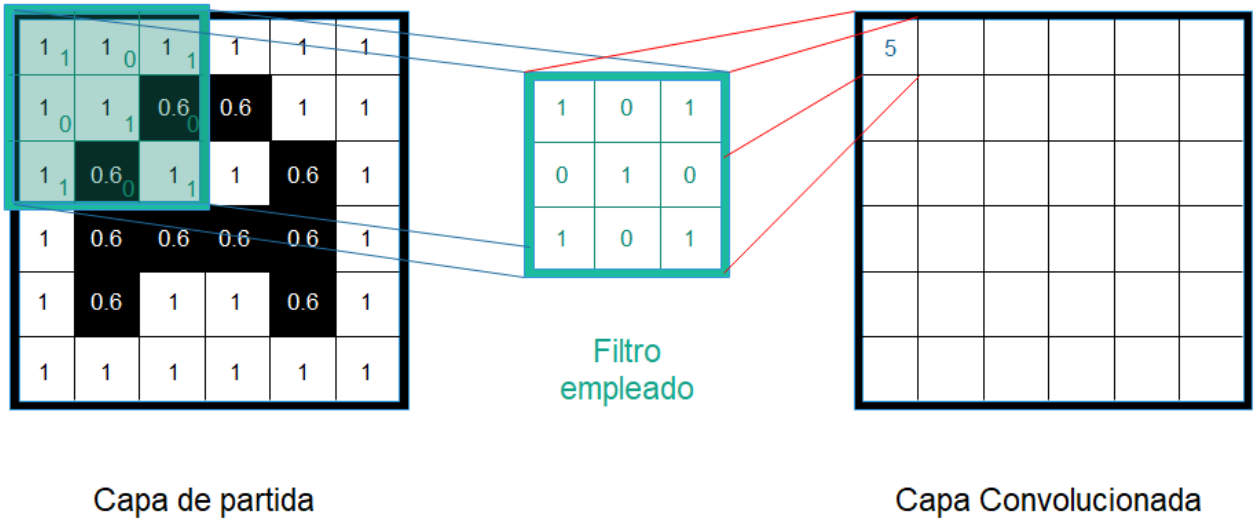

Figura 4. Ejemplo de convolución de una imagen en 2D.

Después de la convolución se suele aplicar una función de activación.

En una red CNN necesitamos reducir el número de neuronas entre una capa oculta y la siguiente. Esto es debido a que, siguiendo el ejemplo anterior, en la primera capa tendríamos 28x28=784 nodos y después de la convolución tendríamos 28x28x32=25088 neuronas. Esto continuaría creciendo exponencialmente lo cual requeriría un procesamiento muy elevado. Por ello, debemos hacer un **submuestreo o *subsampling*** en el cual reducimos el tamaño de nuestras imágenes filtradas teniendo en cuenta que deberán prevalecer las características más importantes que detectó cada filtro.

Normalmente se suele realizar este *subsamling* mediante ***Max-pooling* o agrupación máxima.** Para explicarlo, continuaremos con el ejemplo. Usando un *Max-pooling* de tamaño 2x2, recorreremos cada una de nuestras 32 imágenes de 28x28 pero en lugar de tomar 1 pixel tomaremos 2x2=4 pixeles e iremos quedándonos con el valor más alto de esos 4 pixeles. Haciendo esto la imagen quedaría reducida a la mitad, 14x14. Es decir, pasamos de tener 32 imágenes de 28x28 pixeles, lo que requeriría 25088 neuronas a 32 imágenes de 14x14 pixeles, reduciendo el número de neuronas a 6272 y, teóricamente, siguen almacenando la información más importante para detectar características.

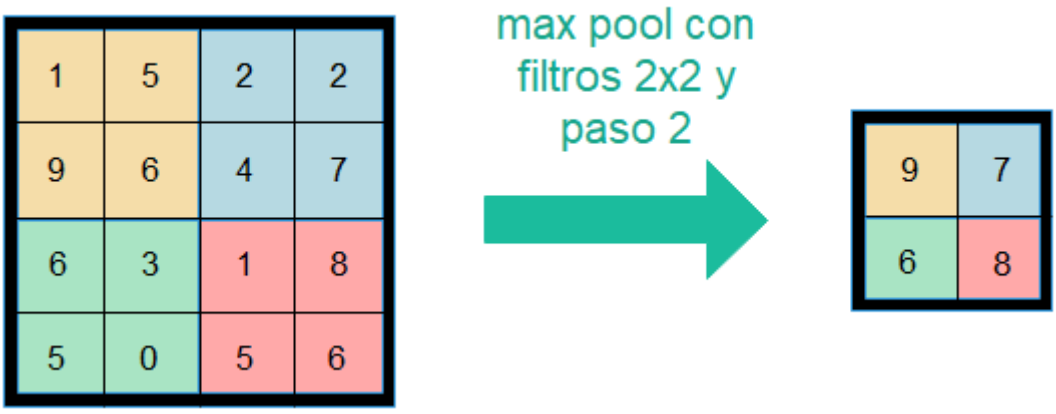

Figura 5. Max pooling de una matriz 4x4.

Resumiendo, los pasos de una capa oculta se pueden resumir en:

1. Entrada
2. Aplico *Kernel*
3. Obtengo el *Feature Mapping*
4. Aplico *Max-Pooling*
5. Obtengo la salida de la convolución

En una CNN introduciremos tantas capas ocultas como consideremos, teniendo en cuenta las características que debe diferenciar, las prestaciones del hardware, la precisión que necesitemos y el tiempo del que disponemos.

Al final de la última oculta tenemos una **capa totalmente conectada o *fully connected layer*.** Está capa se llama así debido a que cada neurona en la primera capa está conectado a todas las neuronas de la segunda capa. Este tipo de capas se utilizan para la clasificación según las características extraídas por las capas exteriores. Está capa suele ser una capa tradicional que contiene una función **softmax** encargada de comprimir los valores de salida a valores entre 0 y 1, generando así una probabilidad para cada una de las etiquetas de clasificación que el modelo pretende predecir.

En la siguiente imagen podemos ver una posible estructura de una CNN:

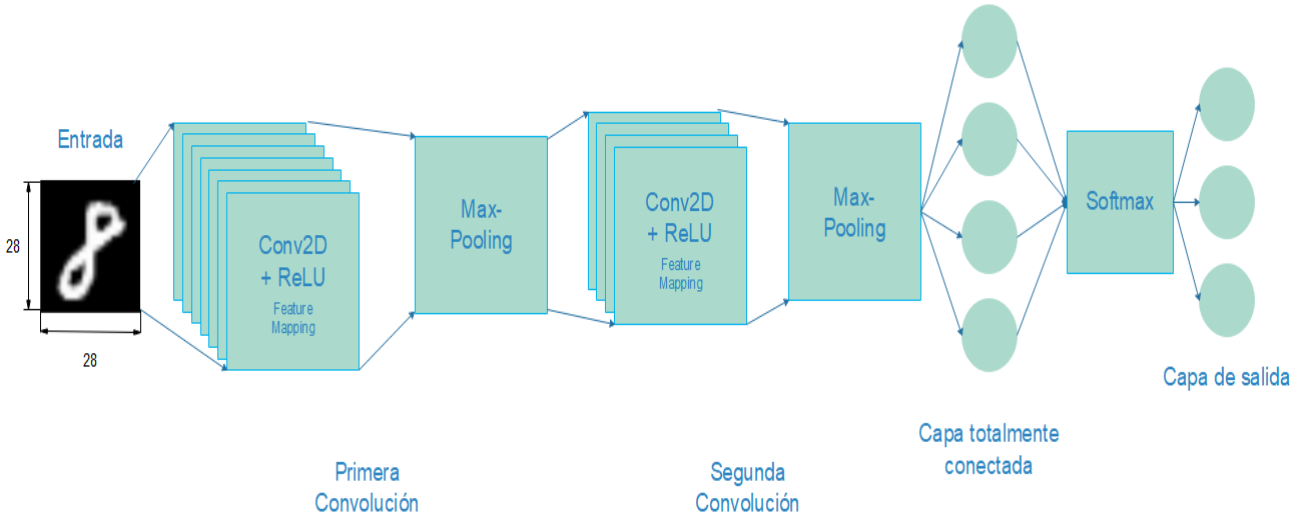

Figura 6. Ejemplo de una arquitectura de CNN.

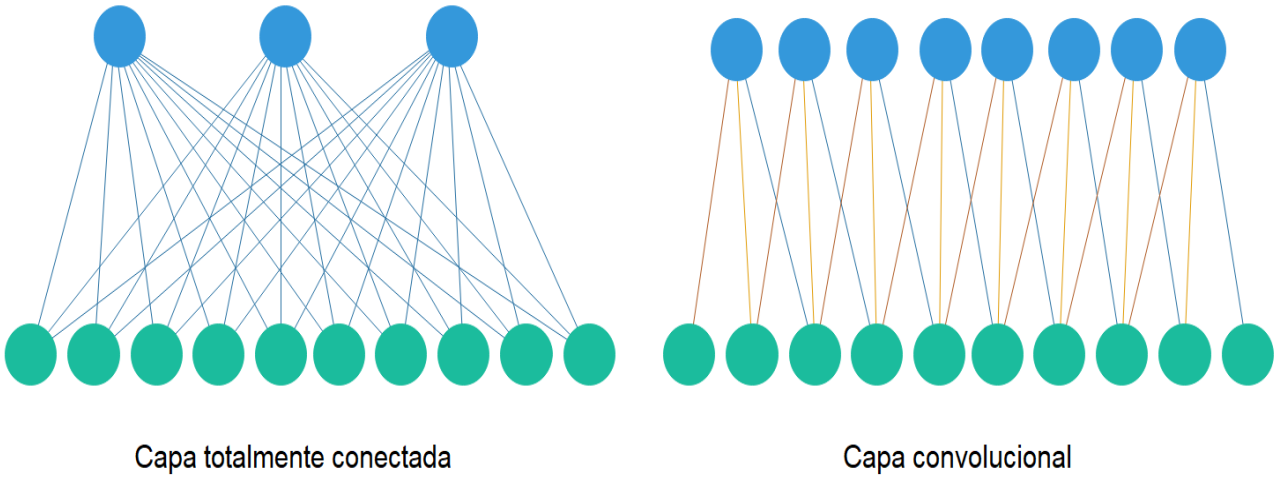

Figura 7. Diferencias de conexiones entre capas totalmente conectadas y capas convolucionales estándar.

Cabe destacar que las salidas al momento del entrenamiento tendrán formato ***one-hot-encoding*** en el cual solo uno de los bits que forman la salida estará a '1' y el resto a '0'.

2.3. Backpropagation (Propagación hacía atrás)

Una CNN aprende mediante el proceso de *Backpropagation*. Este proceso permite mejorar los valores de los pesos de las conexiones entre capas de neuronas mediante la minimización de una función de error que indica la diferencia entre la salida obtenida y la salida deseada. Esta función de error se propaga hacía atrás, partiendo de la capa de salida, hacia todas las neuronas de la capa oculta que contribuyen directamente la salida. Sin embargo las neuronas de la capa oculta solo reciben una fracción de la señal total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa al error total.

La importancia de este proceso consiste en que, a medida que se entrena la red, las neuronas de las capas intermedias se organizan a sí mismas de tal modo que las distintas neuronas aprenden a reconocer distintas características del espacio total de entrada. Después del entrenamiento, cuando se les presente un patrón arbitrario de entrada que contenga ruido o que esté incompleto, las neuronas de la capa oculta de la red responderán con una salida activa si la nueva entrada contiene un patrón que se asemeje a aquella característica que las neuronas individuales hayan aprendido a reconocer durante su entrenamiento.

En la siguiente imagen podemos ver representado el algoritmo de *Backpropagation* en una estructura de redes multicapa, también llamada **perceptrón multicapa**:

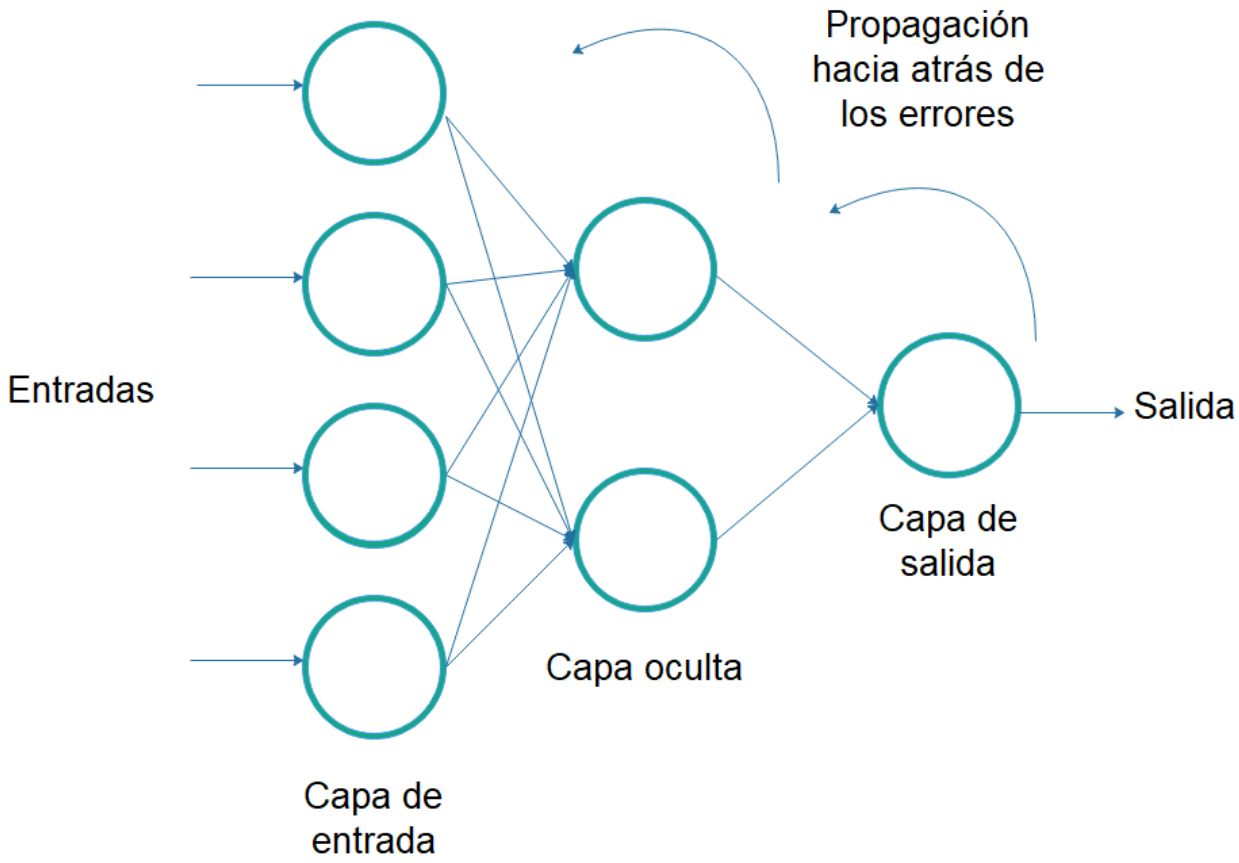

Figura 8. Perceptrón multicapa con propagación hacia atrás.

Como hemos mencionado anteriormente, en una CNN un peso está asociado a cada uno de los pixeles de los distintos *kernels*, lo que hace que se necesite un valor mucho menor de pesos en comparación con redes tradicionales que necesitan un peso por cada neurona.

3. CapsNet (Redes de cápsulas)

3.1. Limitaciones de redes convolucionales

En el aprendizaje profundo, el nivel de activación de una neurona a menudo se interpreta como la probabilidad de detectar una característica específica. Las redes convolucionales son bastante buenas para detectar estas características, pero no son tan buenas para explorar las relaciones espaciales entre dichas características. Esto se debe a que la etiqueta final de estas redes es **invariante punto de vista**, es decir, la red puede detectar de forma errónea una propiedad si detecta la existencia de las características hijas que la forman aunque las propiedades espaciales (posición, colocación y tamaño) de estas características hijas no sean las correctas. Por ejemplo, un modelo simple de una CNN podría extraer características de la nariz, los ojos y la boca correctamente, pero puede que active de manera incorrecta la neurona para detectar una cara. De esta manera, imágenes como la siguiente sería detectada de manera errónea:

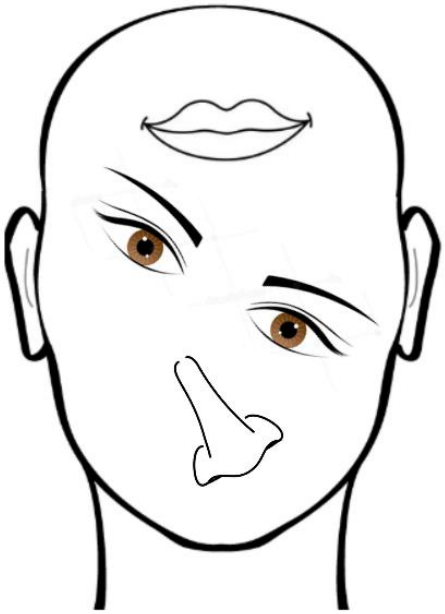

Figura 9. Ejemplo de detección errónea de una cara.

Nos encontramos entonces ante el principal inconveniente de este tipo de redes, el cual incentivó a los investigadores a desarrollar nuevas tecnologías y procesos para solventar dicho problema con un coste computacional reducido.

3.2. Idea de cápsula

Como he mencionado, debemos introducir información sobre el tamaño y la orientación espacial de una característica detectada en nuestro modelo. Podríamos hacerlo agrupando un conjunto de neuronas donde cada una de ellas representa diferentes propiedades de la misma característica del objeto detectado. Es en este punto donde surge la idea de **cápsula**. Al agrupar neuronas, la salida de una cápsula es una matriz de 4x4 (o vector de 16 dimensiones), cuyas entradas representan información como las coordenadas x e y de la característica, el ángulo de rotación del objeto y otras características.

En la siguiente figura podemos ver representado el esquema de una cápsula:

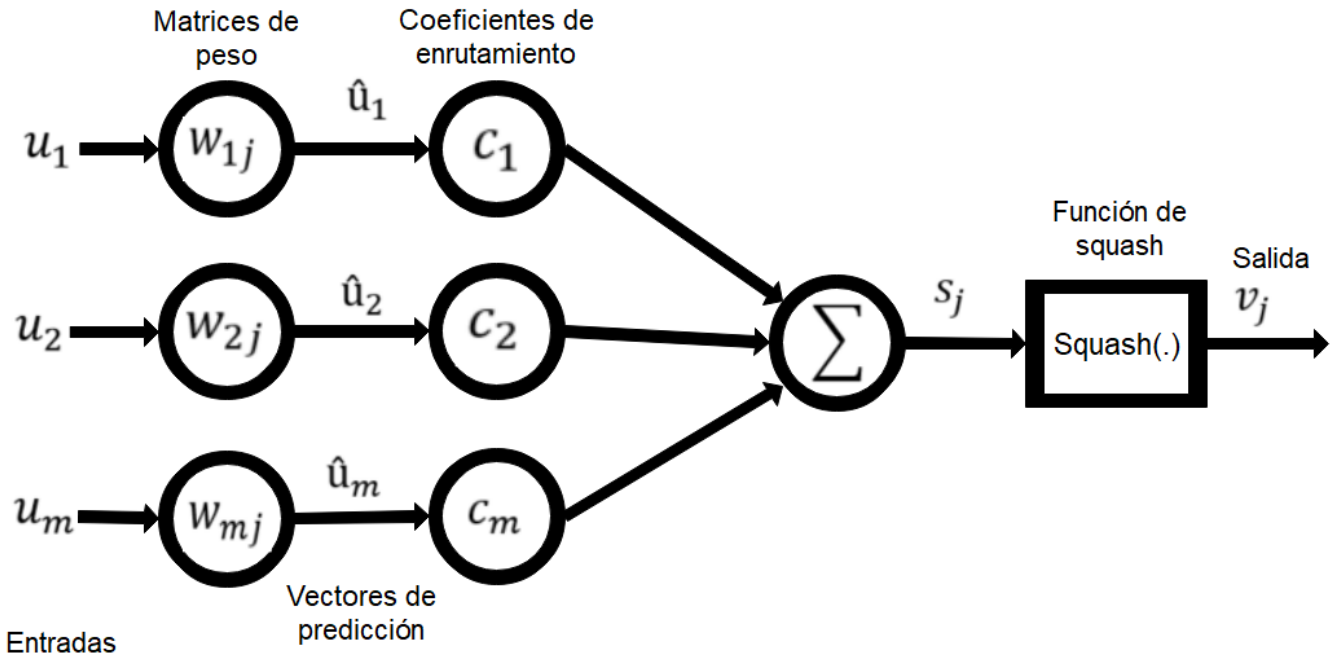

Figura 10. Esquema de una cápsula.

Expliquemos ahora, a partir de la figura anterior, la estructura de una cápsula. Los vectores de entrada u_1, u_2… provienen de las cápsulas de niveles inferiores. Las longitudes de estos vectores codifican las probabilidades de que las cápsulas de nivel inferior detecten sus objetos correspondientes y las direcciones de los vectores codifican algún estado interno de los objetos detectados, como puede ser la rotación o traslación de los mismos.

Estos vectores de entrada se multiplican por matrices de peso (W) correspondientes que codifican relaciones espaciales importantes y otras relaciones entre las entidades de nivel inferior y la entidad de nivel superior. Por ejemplo, W_{2j} puede codificar las relaciones espaciales entre la nariz y la cara. Estas relaciones pueden ser que la cara está centrada alrededor de su nariz, su tamaño es 10 veces el tamaño de la nariz y su orientación en el espacio corresponde a la orientación de la nariz, porque ambas se encuentran en el mismo plano.

Después de la multiplicación por estas matrices, lo que obtenemos es la posición predicha de la característica de nivel superior. En otras palabras, $\hat{u}_2$ nos da información sobre dónde debe estar la cara según la posición detectada de la nariz.

Una vez hecho esto, si las predicciones de características de nivel inferior apuntan a la misma posición y estado de la cara, entonces debe ser una cara allí. Matemáticamente eso obtiene sumando las predicciones y pasándolas por una función de squash que limite los valores para obtener los votos que queremos.

3.3. Dynamic routing based on agreement (Enrutamiento dinámico basado en acuerdo)

3.3.1. Funcionamiento del enrutamiento dinámico

El enrutamiento dinámico es un método de enrutamiento el cual permite a las cápsulas de niveles inferiores decidir a qué cápsula de nivel superior enviará su salida. Podemos explicar esto mejor a través de la siguiente imagen:

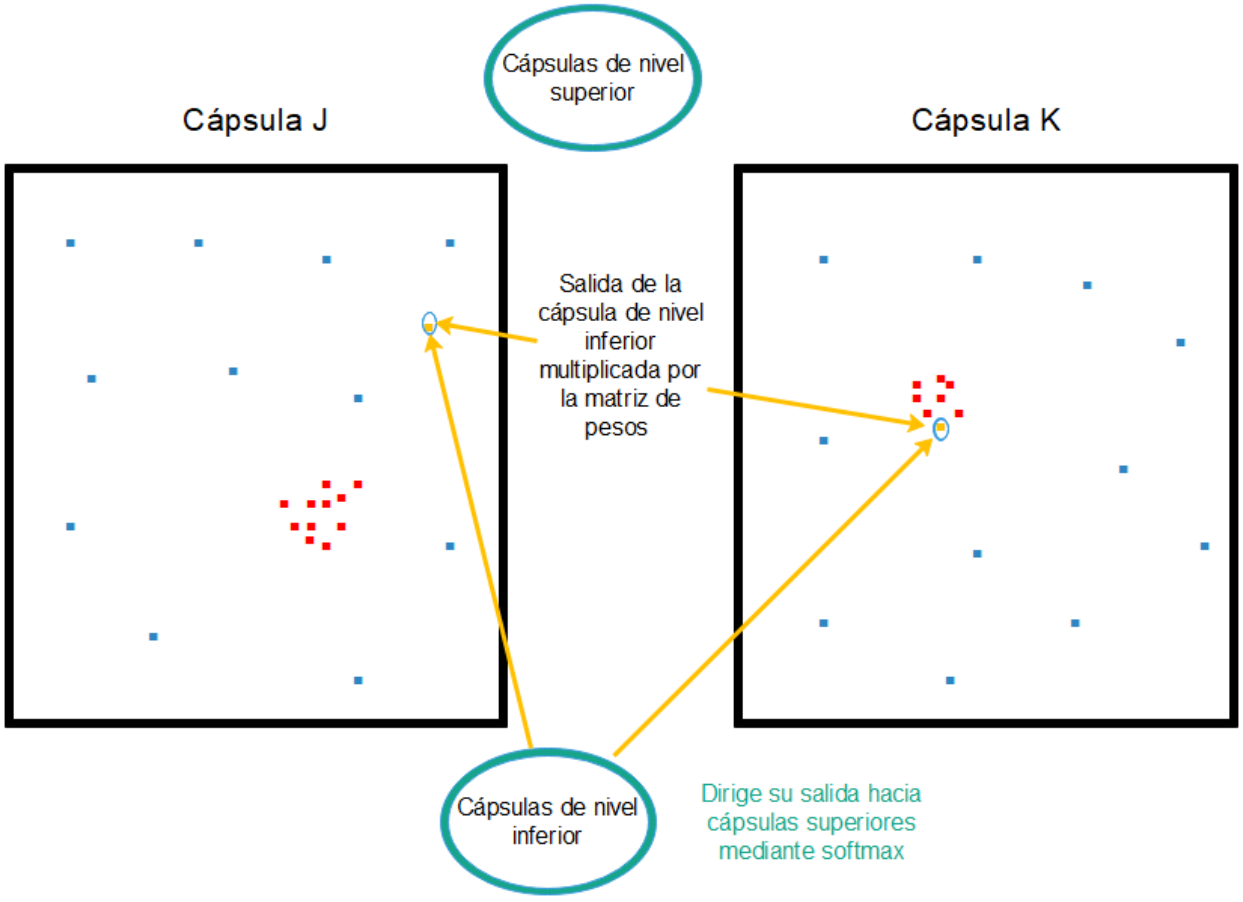

Figura 11. Decisión del enrutamiento dinámico entre dos capas contiguas.

Supongamos que las cápsulas J y K de la imagen son cápsulas de nivel superior conectadas a una cápsula de nivel inferior. Como hemos mencionado, la cápsula de nivel inferior debe de decidir hacia dónde dirigirá su salida. Esta decisión se tomará ajustando los pesos que se multiplicarán por su salida antes de enviarlas a las cápsulas de nivel superior J y K.

Los puntos de la imagen anterior representan las predicciones de enrutamiento. Estas predicciones se obtienen tras la multiplicación de la matriz W por las salidas de las cápsulas de nivel inferior a_i. Los puntos que forman agrupaciones se dibujan en rojo representando el conjunto de predicciones correctas de cada una de las cápsulas de nivel superior (J y K).

La salida de la cápsula inferior al multiplicarse por la matriz de pesos W correspondiente (puntos amarillos), aterriza lejos del grupo rojo de las predicciones correctas en la cápsula izquierda J y, sin embargo, cerca de las predicciones verdaderas del grupo rojo en la cápsula derecha K. La cápsula de

nivel inferior tiene un mecanismo para medir qué cápsula de nivel superior se adapta mejor a sus resultados y ajustará automáticamente su peso de tal manera que el peso correspondiente a la cápsula K será alto, y peso correspondiente a la cápsula J será bajo.

Este proceso es una de las novedades introducidas en las redes de cápsulas. Por contrapartida, el tener este algoritmo de enrutamiento en lugar de hacer *pooling*, hará que nuestro proceso de aprendizaje sea bastante lento, ya que como se puede ver en la siguiente imagen, se tienen que computar más operaciones y además hay que iterar sobre cada cápsula.

3.3.2. Pasos del enrutamiento dinámico por acuerdo

En la siguiente figura tenemos un recorte del documento (Sabour et al., 2017):

Procedure 1 Routing algorithm.

1: **procedure** ROUTING($\hat{\boldsymbol{u}}_{j|i}, r, l$)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$
5: for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6: for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
return $\mathbf{v}_j$

Figura 12. Pasos del enrutamiento dinámico

Analicemos paso a paso este algoritmo:

La primera línea nos dice que este algoritmo toma la capa l de la red de cápsulas, así como sus salidas predichas $\hat{u}_{j|i}$, calculadas tras multiplicar la salida u_i de la cápsula i por la matriz de pesos W_{ij}, y el número de iteraciones de enrutamiento r.

En la segunda línea tenemos el coeficiente b_{ij}. Este nos sirve para guardar los valores temporales de los pesos que se van calculando de manera iterativa. Una vez acabado el procedimiento este valor se almacenará en el coeficiente de enrutamiento c_{ij}. Inicialmente estará inicializado a 0. Además, en esta línea se nos dice que el algoritmo se realizará para todas las cápsulas de la capa l, consideradas cápsulas de nivel inferior, y para todas las cápsulas de la capa $l+1$, consideradas cápsulas de nivel superior.

La tercera línea nos dice que los pasos entre la línea 4 y la línea 7 se repetirán r veces.

El paso de la cuarta línea calcula el valor de todos los coeficientes c_i para una cápsula de nivel inferior i. Utilizamos *softmax* porque así aseguramos que cada peso c_{ij} es un número no negativo comprendido entre 0 y 1 y la suma de todos los coeficientes de enrutamiento s c_{ij} da uno. De esta manera podemos decir que c_{ij} es la probabilidad de que la cápsula i este vinculada con j. Como hemos mencionado, nos encontramos ante una función *softmax* cuya ecuación es la siguiente:

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$$

En la primera iteración el valor de todos los coeficientes c_{ij} será igual, dado que hemos inicializado b_{ij} a cero. Por ejemplo, si tenemos 3 cápsulas de nivel inferior y 2 de nivel superior, entonces todos los coeficientes de enrutamiento c_{ij} serán iguales a 0.5. El estado inicial de estos pesos será el estado de máxima incertidumbre, debido a que las cápsulas de nivel inferior no tienen ni idea de qué cápsulas

de nivel superior se adaptan mejor a su salida. Evidentemente, conforme se repite el proceso, estas distribuciones uniformes cambiarán.

Una vez calculados todos los pesos c_{ij} para las cápsulas de nivel inferior, pasamos a la línea 5 donde trabajamos con las cápsulas de nivel superior. En este paso calculamos una combinación lineal de vectores de predicción $\hat{u}_{j|i}$, ponderándola por los coeficientes de enrutamiento c_{ij} calculados en el paso anterior. Esto crea un vector de salida $s_j = \sum_i c_{ij}\, \hat{u}_{j|i}$

Seguidamente, en la línea 6, pasamos los vectores del paso anterior mediante una función no lineal squash, lo que mantiene la dirección, pero hace que la longitud sea inferior a 1. Este paso crea el vector de salida v_j para todas las cápsulas de nivel superior. La función squash sigue la siguiente fórmula:

$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|}$$

El lado derecho de la ecuación escala el vector de entrada para que tenga la longitud de la unidad sin cambiar su dirección y el lado izquierdo realiza una escala adicional. Recuerde que la longitud del vector de salida puede interpretarse como la probabilidad de que la cápsula detecte una característica dada.

En la siguiente imagen podemos observar la no linealidad de la función squash:

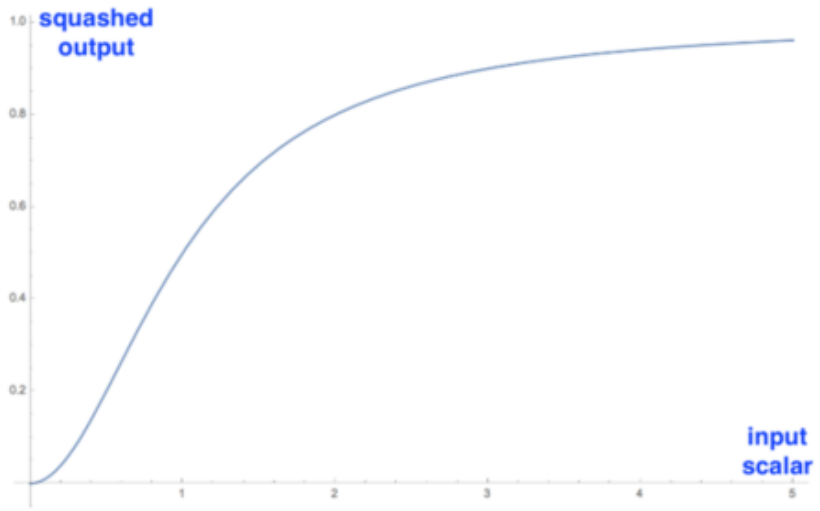

Figura 13. Representación de la función squash.

En esta imagen podemos observar la función aplicada a un vector 1D (a un escalar). Sin embargo, en la aplicación real se tomará un vector como entrada y se obtendrá a la salida otro vector, lo que sería difícil de visualizar.

El paso de la línea 7 actualiza el coeficiente b_{ij} sumándole el producto de la salida de las cápsulas de nivel inferior $\hat{u}_{j|i}$ con la salida de las cápsulas de nivel superior v_j. A este producto se le llama ***agreement*** **(acuerdo)** y mira la similitud entre la entrada de la cápsula y la salida de la misma,

Después de este paso, el algoritmo vuelve a comenzar desde el punto 3 y se repite r veces.

La última línea nos dice que el resultado de este algoritmo es la salida de una cápsula de nivel superior v_j.

Sabiendo esto podemos resumir el funcionamiento de este algoritmo de una manera muy cualitativa diciendo que nos permite calcular el paso hacia delante de la red

Veamos un ejemplo de la actualización de los pesos en la siguiente figura:

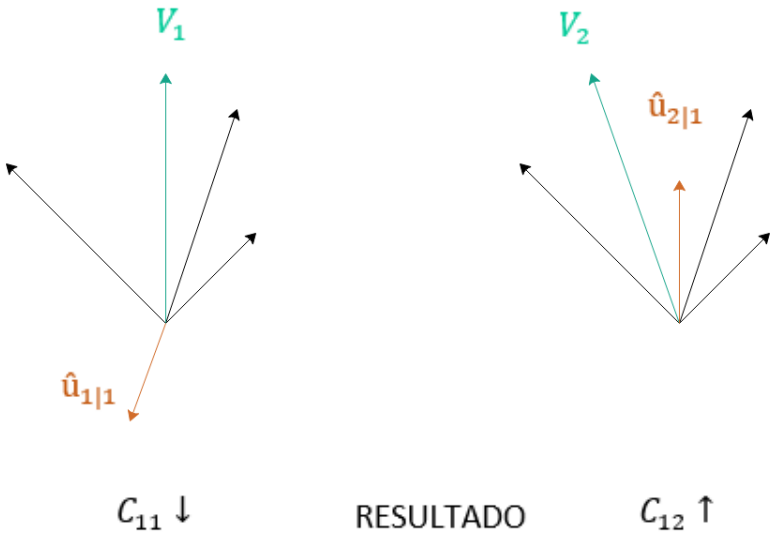

Figura 14. Ejemplo de actualización de los pesos.

Los vectores v_1 y v_2 son las salidas de la cápsula de nivel superior, mientras que $\hat{u}_{1|1}$ y $\hat{u}_{2|1}$ son las salidas predichas de la cápsula de nivel inferior. Como vemos en el ejemplo de la izquierda $\hat{u}_{1|1}$ y v_1 están apuntando a direcciones distintas. Esto hace que su producto sea un número negativo lo que hace que el peso c_{11} disminuya. Sin embargo, en el ejemplo de la derecha pasa lo contrario. v_2 y $\hat{u}_{2|1}$ apuntan a direcciones similares, lo que hace que su peso c_{12} aumente. Esto se repite para todas las cápsulas de nivel superior y para todas las entradas de las cápsulas. El resultado es un conjunto de coeficientes de enrutamiento que se ajustan mejor a las salidas de las cápsulas de nivel superior.

Llegados a este punto puede surgirnos la idea de que un mayor número de iteraciones r puede dar lugar a unos mejores resultados de enrutamiento. Esto no tiene porque ser así, dado que un gran número de iteraciones tiende a sobreajustar los datos, lo que es perjudicial para el funcionamiento de nuestra red.

3.3.3. Margin Loss (Margen de pérdida)

Usamos la longitud del vector de predicción v_j para representar la probabilidad de que exista la entidad de una cápsula. Nuestro objetivo es que la clase de dígitos k tenga un vector de predicción largo si ese y solo si ese digito está presente en la imagen. Para permitir la detección de múltiples dígitos de forma simultánea, usamos márgenes de pérdida separados L_k para cada clase de dígitos k:

$$L_k = T_k \max(0, m^+ - \|v_k\|^2) + \lambda(1 - T_k)\max(0, \|v_k\| - m^-)^2$$

T_k será igual a uno si un digito de la clase k está presente y $m^+ = 0.9$ y $m^- = 0.1$. λ, por defecto 0.5, es una ponderación descendente que evita que el aprendizaje inicial reduzca los vectores de actividad de todas las clases. El margen de pérdida total es la suma de todos los márgenes de pérdidas.

3.4. Arquitectura

En este apartado vamos a explicar la arquitectura que Hinton empleo en sus desarrollos de *CapsNet* en el documento (Sabour et al., 2017). Para ello, tomamos imágenes del conjunto de datos de MNIST a modo de ejemplo e iremos explicando parte por parte toda la estructura de la red de cápsulas. En la siguiente imagen podemos ver dicha estructura:

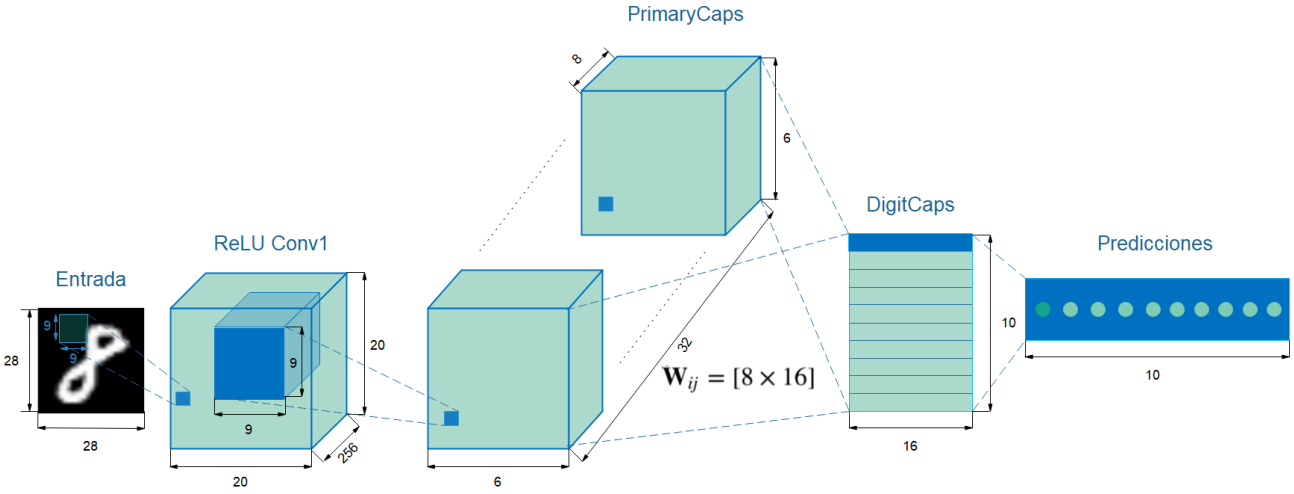

Figura 15. Arquitectura de CapsNet.

Esta red comienza con una primera capa que es un detector de características basado en convolución. Esta capa convolucional (**Conv1**) aplica 256 *kernels* (núcleos) de 9x9 píxeles a cada canal de color a cada imagen de entrada. Por tanto, para imágenes en blanco y negro (con un solo canal de color), la forma de los filtros es de 9x9x1. Para la convolución se considera un paso de 1 y sin relleno, lo que reduce el tamaño de las imágenes originales de MNIST de 28x28 píxeles a 20x20 píxeles ($\left\lfloor \frac{28-9}{1} \right\rfloor + 1 = 20$). Se han elegido 256 filtros para esta capa, por tanto, esta capa tiene una forma de 20x20x256. Antes de pasar a la siguiente capa se pasa por una función de activación ReLu. Finalmente, se obtienen 256 canales.

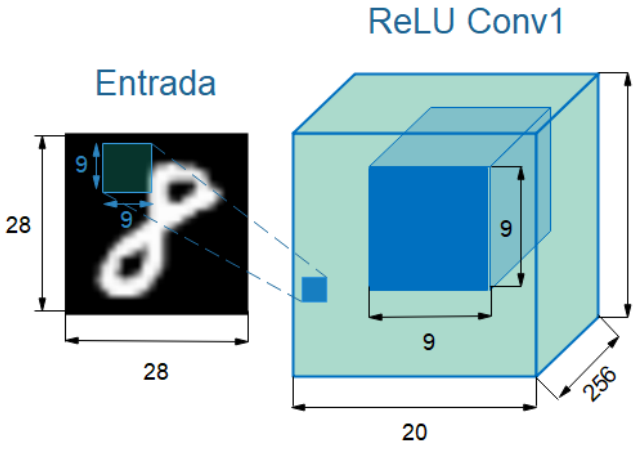

Figura 16. Primera capa de detección de características.

En la segunda etapa de esta red se vuelve a realizar una detección de características. Sin embargo, en esta etapa se dividen los 256 mapas de características anteriores en 32 canales de cápsulas de 8 dimensiones, agrupando neuronas de 8 en 8. A esta etapa se le llama ***PrimaryCaps*** y usa *kernels* de

9x9x256 con un paso de 2 y sin relleno reduciendo así la dimensión espacial de 20x20 a 6x6 ($\left\lfloor\frac{20-9}{2}\right\rfloor + 1 = 6$). Finalmente, *PrimaryCaps* tiene 32x6x6=1152 vectores de longitud 8.

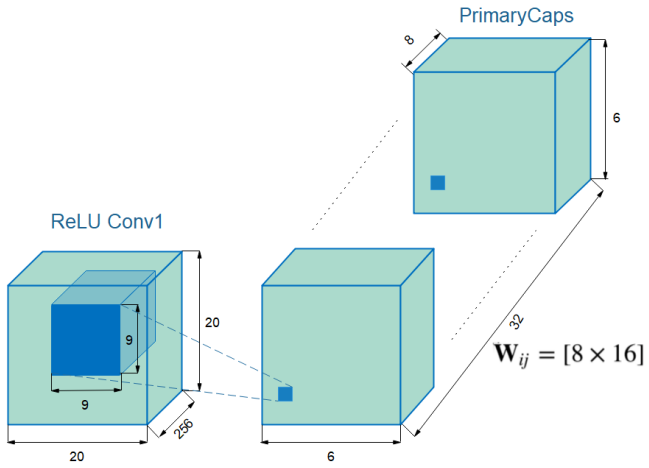

Figura 17. Segunda capa de detección de características y mapeado a cápsulas.

La tercera etapa recibe el nombre de *DigitCaps* y aplica una matriz de transformación W_{ij} con forma 16x8 para convertir las 1152 cápsulas de 8-D en una cápsula 16-D para cada clase j. En nuestro caso nuestras clases j son 10 (los 10 dígitos que tiene MNIST). Por tanto, la forma final de ***DigitCaps*** es 10x16.

Para realizar esto, se emplea enrutamiento dinámico por acuerdo. Si multiplicamos los 1152 vectores u_i por la matriz W_{ij} obtenemos los vectores de predicción $\hat{u}_{j|i}$. Necesitamos un vector de predicción $\hat{u}_{j|i}$ por cada *PrimaryCaps* y por cada *DigitCaps*, por lo que necesitaremos 1152x10=11520 vectores y sus correspondientes 11520 matrices de transformación W_{ij}. Como se ha explicado en el apartado 3.3 mediante la suma ponderada de todos los vectores $\hat{u}_{j|i}$ y tras el paso de la función de squash obtenemos 10 DigitsCaps de 16-D.

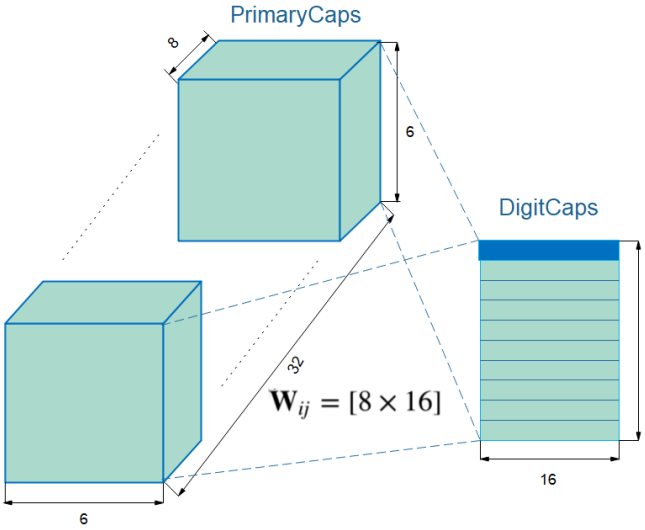

Figura 18. Mapeado entre PrimaryCaps y DigitCaps.

La última capa es solo un cálculo de la longitud de los vectores de dígitos v_j, realizado a partir de su norma, por lo que hay 10 valores por imagen en los que la activación más alta determina el dígito actual.

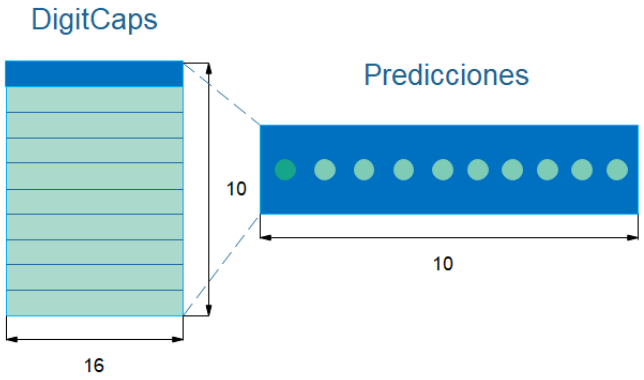

Figura 19. Obtención de predicciones en CapsNet.

3.5. Reconstrucción de imágenes

Durante el entrenamiento, enmascaramos todo menos el vector de actividad de la cápsula de dígitos correcta. Los votos v_j finalmente obtenidos a la salida de *DigitCaps* se conectan a un **decodificador** formado por 3 capas totalmente conectadas. De esta forma, podemos regenerar la imagen original. Además de esto, esta técnica actúa como un método de regularización y es útil para facilitar a las cápsulas de dígitos a codificar los parámetros de instanciación del dígito de entrada.

El objetivo es minimizar las pérdidas de reconstrucción, las cuales se calculan como la suma de las diferencias al cuadrado entre las intensidades de los píxeles de entrada y la salida del decodificador. Esta función se multiplica por un **factor de regularización**, típicamente con valor igual a 0.0005, para que no domine el margen de pérdidas. Posteriormente, el resultado de lo anterior se agrega a la función de pérdidas.

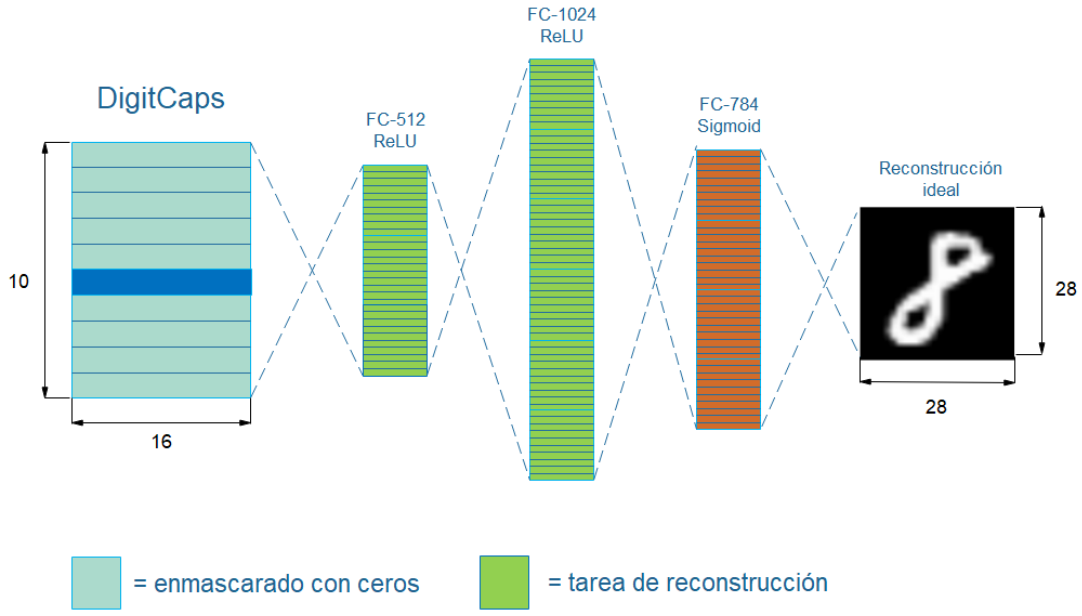

Figura 20. Reconstrucción de la imagen de entrada a partir de DigitCaps.

4. Matrix CapsNet (Red de cápsulas matriciales)

4.1. Introducción

Como he mencionado, la invención de las redes de cápsulas ha sido un paso importante para el mundo de la inteligencia artificial. En su afán por seguir mejorando, multitud de desarrolladores han continuado investigando las redes de cápsulas. En 2018, Geoffrey Hinton, Sara Sabour y Nicholas Frosst publicaron un documento **(Hinton et al., 2018)** en el que exponían sus últimos avances, las redes de cápsulas matriciales. En este apartado, me dispongo a explicar dichas redes.

4.2. Funcionamiento

Una matriz de cápsulas captura la activación de manera similar a como lo hace una neurona y además, captura una **matriz de pose o matriz de posición 4x4**. Esta matriz define la traslación y la rotación de un objeto que es equivalente al cambio del punto de vista de dicho objeto. En las redes de cápsulas originales usábamos un vector como salida de una cápsula en lugar de una matriz de pose y un valor de activación, lo que hace de esto la principal diferencia entre ambos tipos de redes.

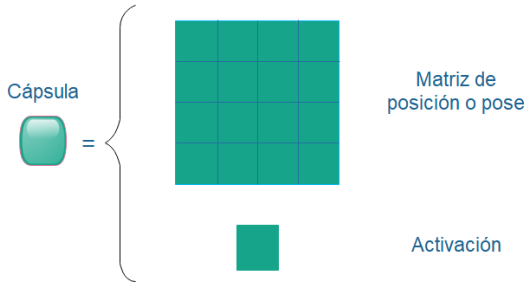

Figura 21. Estructura cápsula en MatCaps.

Otra gran diferencia entre las redes de cápsulas y las redes de cápsulas matriciales es el tipo de algoritmo que utilizan para realizar el enrutamiento. Mientras que las primeras usan el enrutamiento dinámico, ya explicado anteriormente en este documento, las segundas emplean **enrutamiento por acuerdo EM**. Este será explicado a continuación.

4.3. EM for routing-by-agreement (Enrutamiento por acuerdo EM)

El principal objetivo del enrutamiento EM (*Expectation Maximization*) es agrupar cápsulas para formar una **relación parte-todo** usando una técnica de agrupamiento (EM). Podemos usar clústeres de EM para agrupar puntos de datos en **distribuciones gaussianas** $G_1 = N(\mu_1, \sigma_1^2)$ y $G_2 = N(\mu_2, \sigma_2^2)$. Seguidamente representamos los puntos de datos con la distribución gaussiana correspondiente.

Volviendo al ejemplo de detección facial, cada una de las cápsulas de detección de boca, ojos y nariz en la capa inferior hace predicciones (**votos**) en las matrices de postura de sus posibles cápsulas originales. Cada voto es un valor predicho para la matriz de pose de una cápsula principal, y se calcula multiplicando su propia matriz de pose M con una **matriz de transformación invariante al punto de vista (*viewpoint invariant transformation matrix*)** W que aprendemos de los datos de entrenamiento. $v = MW$

Aplicamos el enrutamiento EM para agrupar cápsulas formando una cápsula principal en tiempo de ejecución. Es decir, si las cápsulas de la nariz, boca y cara votan un valor de matriz de pose similar, las agrupamos para formar una cápsula principal (cápsula de la cara).

4.3.1. Funcionamiento del enrutamiento por acuerdo EM

Antes de comprender el enrutamiento EM debemos comprender el algoritmo EM. Un modelo de mezcla Gaussiana agrupa los puntos de datos en una mezcla de distribuciones gaussianas descritas por una **media μ** y una **desviación típica σ**. Seguidamente agrupamos los puntos de datos en conjuntos los cuales cada uno de ellos es representado con una μ y una σ.

Para un modelo de mezcla Gaussiano, empezamos con una iniciación aleatoria de los clúster $G_1 = N(\mu_1, \sigma_1^2)$ y $G_2 = N(\mu_2, \sigma_2^2)$. El algoritmo EM trata de ajustar los puntos de entrenamiento de G_1 y G_2 y vuelve a calcular μ y σ para los *clúster* basándose en una distribución Gaussiana.

La probabilidad de que un punto x este en un *clúster* G_1es la siguiente:

$$P(x|G_1) = \frac{1}{\sigma_1\sqrt{2\pi}} e^{-(x-\mu_1)^2/2\sigma_1^2}$$

En cada iteración, empezamos con dos distribuciones Gaussianas que recalcularemos sus μ y σ en función de los datos.

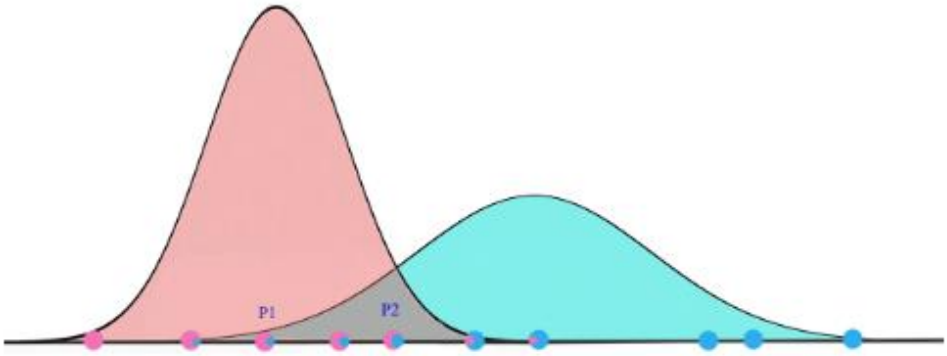

Figura 22. Ejemplo de distribuciones Gaussianas iniciales.

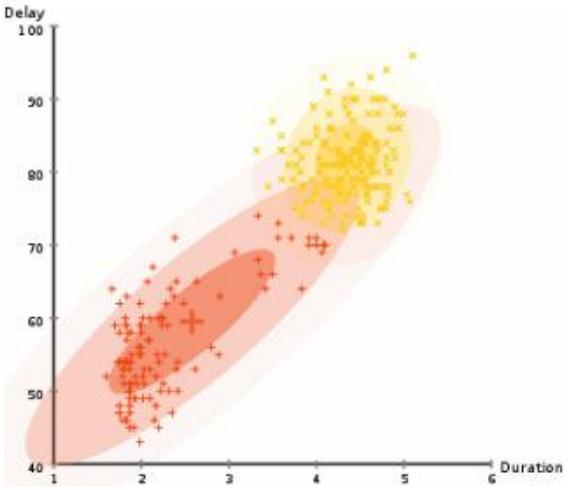

Figura 23. Inicialización de distribuciones Gaussianas sobre nube de puntos formada por los datos de entrada.

Eventualmente, convergeremos a dos distribuciones gaussianas que maximizan la separación de los puntos de datos observados.

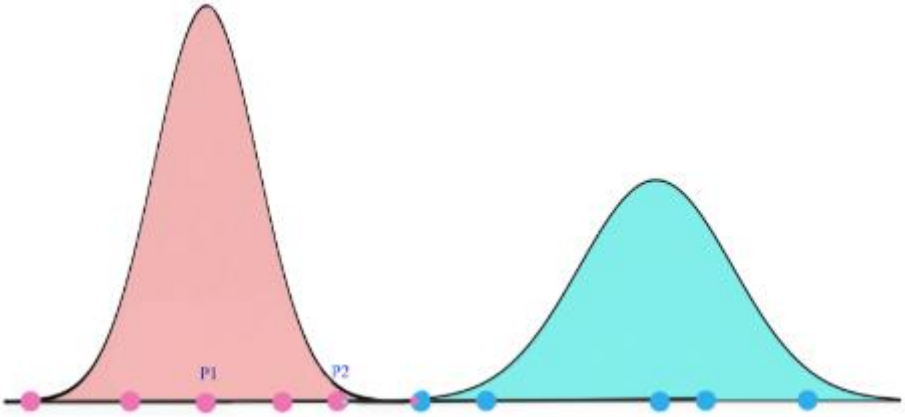

Figura 24. Ejemplo de distribuciones Gaussianas que maximizan la separación.

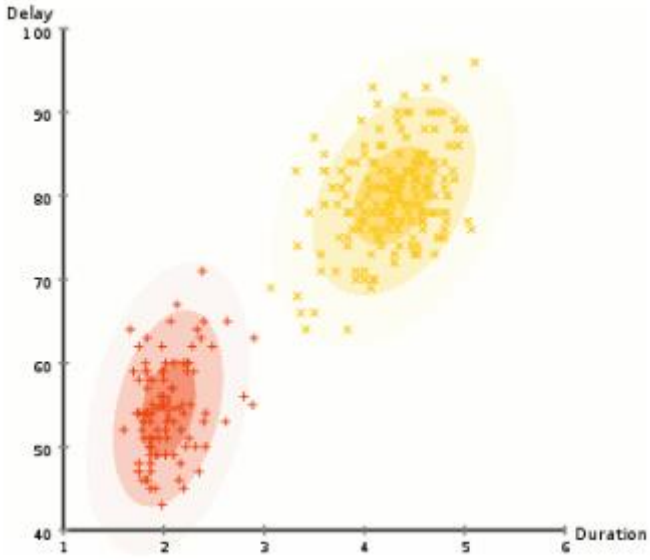

Figura 25. Convergencia de las distribuciones Gaussianas tras la adaptación de su media y desviación típica.

Como hemos mencionado anteriormente, una característica de nivel superior se detecta buscando un acuerdo entre los votos de las cápsulas de nivel inferior. Un voto $\boldsymbol{v_{ij}}$ para la cápsula madre j se calcula multiplicando la matriz de pose M_i de las cápsulas hija i con una matriz de transformación invariable al punto de vista $\boldsymbol{W_{ij}}$.

Si el voto v_{ij} está próximo a otros votos de otras cápsulas hijas, la probabilidad de que esa cápsula hija i se agrupe con una cápsula madre j será mayor. W_{ij} aprende discriminativamente a través de una **función de costo** y la **propagación hacia atrás**, explicada anteriormente.

En la siguiente imagen podemos observar la visualización de enrutamiento por acuerdo con las cápsulas matriciales. Agrupamos cápsulas con votos similares ($v_{ij} \approx v_{tj} \Rightarrow M_i W_{ij} \approx M_t W_{tj}$):

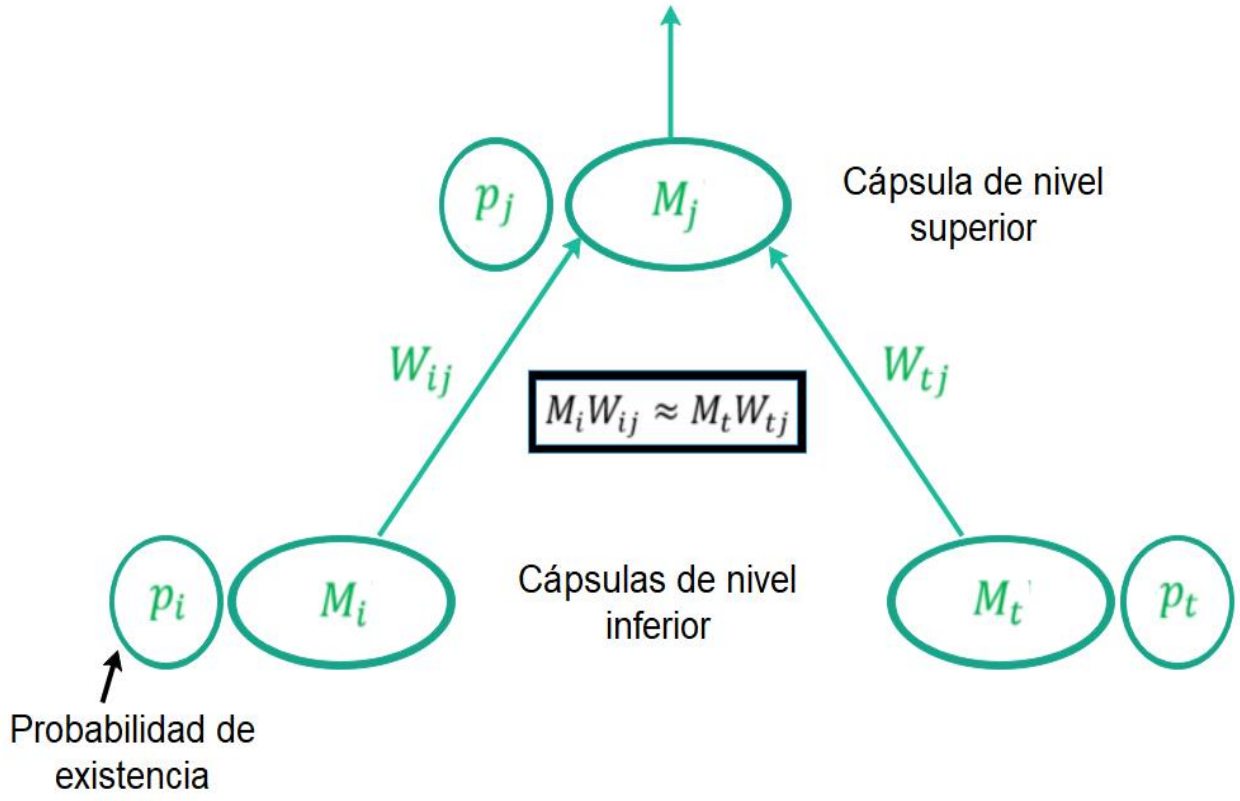

Figura 26. Esquema del enrutamiento por acuerdo EM.

En el caso de que cambie el punto de vista, las matrices de pose y los votos cambian de forma coordinada. Por tanto, los votos cambiaran. Sin embargo, como hemos mencionado, el enrutamiento EM se basa en la proximidad de estos votos entre si y podría agrupar las mismas cápsulas. Por este motivo, las matrices de transformación son las mismas para cualquier punto de vista de los objetos (invariante al punto de vista). Solo necesitaremos un conjunto de matrices de transformación y una cápsula principal para detectar diferentes posiciones de objetos.

Además, el enrutamiento por acuerdo calcula las **probabilidades de asignación r_{ij}**. Esta parámetro nos dice la probabilidad de que una cápsula de nivel inferior i este asociada a una cápsula de nivel superior j en tiempo de ejecución. Explicaremos esto con un ejemplo. Una cápsula que detecte dedos de una mano tendrá una probabilidad de asignación nula con la cápsula que detecta una cara. Sin embargo, tendrá una probabilidad muy alta con la cápsula de la mano.

Como hemos mencionado, en el enrutamiento EM, representamos puntos de datos mediante una distribución gaussiana. También modelamos la matriz de pose de la cápsula principal. Para ello, esta matriz es de dimensiones 4x4, por tanto, necesitamos 16 componentes. Estos componentes los modelaremos mediante un modelo gaussiano de 16 μ y 16 σ, donde cada μ representa un componente de la matriz de pose.

4.3.2. Decisión de enrutamiento

Tenemos la opción de decidir si una cápsula de nivel superior se activa o no. Para ello hay que tener en cuenta dos cosas:

- Si la activamos debemos pagar un **costo fijo de $-\boldsymbol{\beta_u}$ por punto de datos** para describir las poses de todas las cápsulas de nivel inferior que se asignan a la cápsula de nivel superior. Este costo es el negativo del logaritmo de la densidad de probabilidad del punto de datos bajo una distribución uniforme incorrecta previa. Para las asignaciones fraccionarias, pagamos esa fracción del costo fijo.
- Si activamos la cápsula de nivel superior, debemos pagar un **costo fijo de $-\boldsymbol{\beta_a}$ por codificar su media y varianza** y por el hecho de que está activa y luego pagar costos adicionales, prorrateados por las probabilidades de asignación r_{ij}, para describir las discrepancias entre las medias μ de nivel inferior y los valores predichos para ellas, cuando se usa la media de la cápsula de nivel superior para predecirlas a través de la inversa de la matriz de transformación. Una manera mucho más simple de calcular el costo de describir un punto de datos es usar el negativo del logaritmo de la densidad de probabilidad del voto de ese punto de datos bajo la distribución gaussiana ajustada por cualquier cápsula de nivel superior asignada a los objetivos. Esto no es del todo correcto, pero lo usamos porque es una aproximación que requiere mucho menos cómputo.

La diferencia de los costos anteriores se pasa a través de la función logística en cada iteración para determinar la probabilidad de activación de la cápsula de nivel superior a_j. Usando esto podemos calcular el costo de activar una cápsula.

Matemáticamente este costo es $-\ln(P_{i|j}^h)$, cuya fórmula es:

$$P_{i|j}^h = \frac{1}{\sqrt{2\pi(\sigma_j^h)^2}} e^{-\frac{(v_{ij}^h-\mu_j^h)^2}{2(\sigma_j^h)^2}}$$

$$cost_j^h = -\ln(P_{i|j}^h) = \frac{(v_{ij}^h-\mu_j^h)^2}{2(\sigma_j^h)^2} + \ln(\sigma_j^h) + \frac{\ln(2\pi)}{2}$$

Donde $P_{i|j}^h$ es la densidad de probabilidad del componente h del voto vectorizado v_{ij} bajo el modelo gaussiano de la cápsula superior *j* para la dimensión h que tiene varianza $(\sigma_j^h)^2$ y media μ_j^h donde μ_j es la versión vectorizada de la matriz de pose de la cápsula *j*, M_j

A mayor costo, más difícil es que esta cápsula se active, debido a que los votos no coinciden con la distribución Gaussiana de esa cápsula.

La fórmula anterior sería así siempre y cuando las cápsulas de niveles inferiores estén vinculadas por igual con la cápsula j. Como esto no suele ser así, prorrateamos el costo con las probabilidades de asignación r_{ij}.

$$cost_j^h = \sum_i -r_{ij}\ln(P_{i|j}^h) = \sum_i -r_{ij}\left(\frac{(v_{ij}^h-\mu_j^h)^2}{2(\sigma_j^h)^2} + \ln(\sigma_j^h) + \frac{\ln(2\pi)}{2}\right)$$

Para calcular la **activación** de la cápsula j usamos la siguiente ecuación:

$$a_j = sigmoid(\lambda(\beta_a - \beta_u \sum_i r_{ij} - \sum_h cost_j^h))$$

Como hemos explicado, β_a es un costo igual para todas las cápsulas y $\boldsymbol{\lambda}$ **es la temperatura inversa** que será fijada en el enrutamiento y modifica la inclinación de la curva de la función sigmoide. Esto nos es útil para afinar mejor r_{ij}. β_a y β_u aprenderán discriminativamente.

4.3.3. Pasos del enrutamiento EM

A continuación, nos disponemos a explicar paso a paso el algoritmo de enrutamiento EM a partir de los siguientes recortes del documento **(Hinton et al., 2018)**:

1: **procedure** EM ROUTING(a, V)
2: $\forall i \in \Omega_L, j \in \Omega_{L+1}$: $R_{ij} \leftarrow 1/|\Omega_{L+1}|$
3: **for** t iterations **do**
4: $\forall j \in \Omega_{L+1}$: M-STEP($a$, R, V, j)
5: $\forall i \in \Omega_L$: E-STEP(μ, σ, a, V, i)
return a, M

Figura 27. Pasos del enrutamiento EM.

1. La primera línea nos dice que el algoritmo tomará como entrada las activaciones **a** y los votos **v** de las cápsulas de nivel inferior L.
2. La segunda línea nos dice que para toda combinación de cápsulas de nivel inferior con cápsulas de nivel superior, inicializaremos una probabilidad de asignación $\boldsymbol{r_{ij}}$ de manera uniforme como uno partido del número de cápsulas de la capa superior $\boldsymbol{\Omega_{L+1}}$
3. La tercera línea nos dice que los pasos de las líneas 4 y 5 se repetirán **t** veces. Esta t es el número de iteraciones del algoritmo de enrutamiento EM, suelen ser 2 o 3.
4. La cuarta línea nos indica que debemos realizar el paso M para todas las cápsulas del nivel superior. Este paso se explicará un poco más adelante.
5. La quinta línea nos indica que debemos realizar el paso E para todas las cápsulas del nivel inferior. Este paso se explicará un poco más adelante.
6. La última línea nos indica que el algoritmo nos dará como salida la activación **a** actualizada y la matriz de pose **M**.

Expliquemos ahora el paso M.

1: **procedure** M-STEP(a, R, V, j) ▷ for one higher-level capsule, j
2: $\forall i \in \Omega_L$: $R_{ij} \leftarrow R_{ij} * a_i$
3: $\forall h$: $\mu_j^h \leftarrow \frac{\sum_i R_{ij} V_{ij}^h}{\sum_i R_{ij}}$
4: $\forall h$: $(\sigma_j^h)^2 \leftarrow \frac{\sum_i R_{ij} (V_{ij}^h - \mu_j^h)^2}{\sum_i R_{ij}}$
5: $cost^h \leftarrow \left(\beta_u + log(\sigma_j^h)\right) \sum_i R_{ij}$
6: $a_j \leftarrow logistic(\lambda(\beta_a - \sum_h cost^h))$

Figura 28. Paso M del enrutamiento EM.

1. Este paso nos es útil dado que nos permite calcular el modelo gaussiano (μ, σ) y la activación de las cápsulas de nivel superior, introduciendo como parámetros de entrada las activaciones

de las cápsulas de nivel inferior **a**, los votos **V**, la probabilidad de asignación **R** y la cápsula de nivel superior **j** a la que queremos calcularle el modelo.

2. En la línea 2, recalculamos la probabilidad de asignación $\boldsymbol{r_{ij}}$ multiplicando el valor que tenía anteriormente por la activación de la cápsula de nivel inferior $\boldsymbol{a_i}$.
3. En la línea 3, calculamos la media del modelo de la cápsula de nivel superior para cada dimensión $\boldsymbol{\mu_j^h}$.
4. En la cuarta línea, calculamos la varianza del modelo de la cápsula de nivel superior para cada dimensión $(\boldsymbol{\sigma_j^h})^{\boldsymbol{2}}$.
5. En la quinta línea, calculamos el coste de que una cápsula de nivel superior sea activada por una de nivel inferior.
6. En la última línea, calculamos la activación de la cápsula de nivel superior $\boldsymbol{a_j}$.

Ahora, voy a explicar el paso E:

1: **procedure** E-STEP(μ, σ, a, V, i) ▷ for one lower-level capsule, i

2: $\forall j \in \Omega_{L+1}$: $p_j \leftarrow \frac{1}{\sqrt{\prod_h^H 2\pi(\sigma_j^h)^2}} \exp\left(-\sum_h^H \frac{(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right)$

3: $\forall j \in \Omega_{L+1}$: $R_{ij} \leftarrow \frac{a_j p_j}{\sum_{k \in \Omega_{L+1}} a_k p_k}$

Figura 29. Paso E del enrutamiento EM.

Este paso toma como entrada el modelo gaussiano (μ, σ) y las activaciones $\boldsymbol{a_j}$ calculadas en el paso M y los votos y la cápsula de nivel inferior al que se le va a aplicar dicho paso.

Lo que hace dicho paso es recalcular la probabilidad de asignación $\boldsymbol{r_{ij}}$ basándose en el nuevo modelo y $\boldsymbol{a_j}$. La asignación se incrementa si el voto está más cerca de la μ del modelo gaussiano actualizado.

Finalmente, usamos el $\boldsymbol{a_j}$ desde la última llamada de m-step en las iteraciones como la activación de la cápsula de salida j y damos forma a los 16 μ para formar la matriz de pose 4x4.

4.3.1. Función de pérdida de propagación o Spread Loss

Con el fin de hacer el entrenamiento más sensible a la inicialización de los hiperparámetros del modelo, maximizamos el *gap* entre la activación de la clase objetivo $\boldsymbol{a_t}$ y la activación de las otras clases. Además, necesitamos entrenar la matriz de transformación **W** y los parámetros $\boldsymbol{\beta_v}$ y $\boldsymbol{\beta_\alpha}$. Para ello, empleamos la función de pérdidas de propagación o *Spread Loss*:

$$L_i = (\max(0, \mathrm{m} - (a_t - a_i)))^2$$

Donde $\boldsymbol{L_i}$ son las pérdidas de una clase hija i, $\boldsymbol{a_t}$ es la activación de la clase objetivo con una etiqueta verdadera, $\boldsymbol{a_i}$ es la activación de la clase i y **m** es el margen.

Las pérdidas totales son $L_i = \sum_{i \neq t} L_i$

Si la diferencia entre la etiqueta verdadera, a_t, y la activación de la clase i, a_i, es menor que el margen m, penalizamos la red con unas pérdidas de $\mathrm{m} - (a_t - a_i)^2$. En caso contrario, no la penalizamos. De esta manera, aumentamos el margen entre la activación de la clase objetivo y la activación del resto de clases. El valor del margen m comienza con un valor pequeño (0.2) para hacer la red menos sensible a la inicialización de los hiperparámetros del modelo y aumenta linealmente en 0.1 después de cada entrenamiento de época. M dejará de crecer después de alcanzar el máximo de 0.9. Comenzar en un

margen más bajo ayuda al entrenamiento a evitar demasiadas cápsulas muertas durante la fase temprana.

4.4. Arquitectura

En este apartado vamos a explicar la estructura de las redes de cápsulas matriciales especificada en el documento **(Hinton et al., 2018)**. Para ello, al igual que hemos hecho en el apartado de las redes de cápsulas vectoriales, vamos a utilizar el conjunto de datos de MNIST para apoyar nuestra explicación. En la siguiente imagen podemos observar la arquitectura de estas redes:

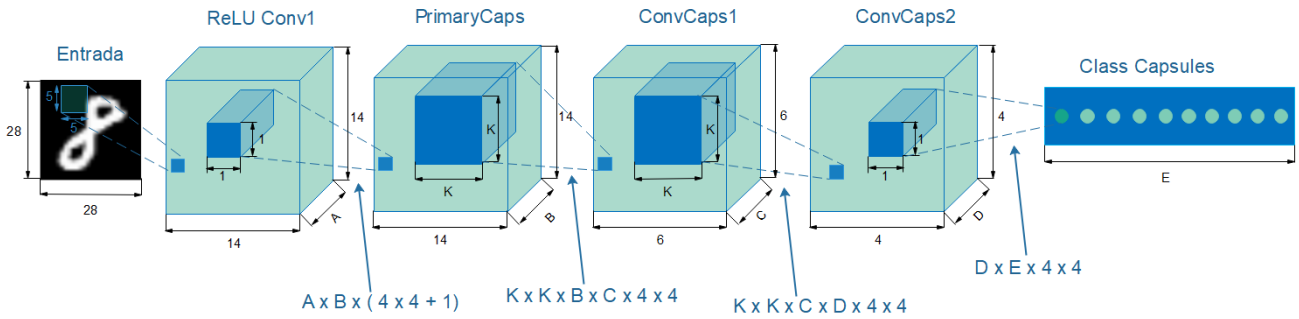

Figura 30. Arquitectura Matrix Capsules.

Comenzamos al igual que en el caso de *CapsNet* con una primera capa que es un detector de características basado en convolución. Esta capa convolucional (**Conv1**) aplica 32 *kernels* (núcleos) de 5x5 píxeles a cada canal de color a cada imagen de entrada. Por tanto, para imágenes en blanco y negro (como MNIST), la forma de los filtros es de 5x5x1. Para la convolución se considera un paso de 2 y con relleno, lo que reduce el tamaño de las imágenes originales de MNIST de 28x28 píxeles a 14x14 píxeles ($\left\lfloor \frac{28+2x2-5}{2} + 1 \right\rfloor = 14$). Se han elegido 32 filtros para esta capa (A=32), por tanto, esta capa tiene a su salida 32 canales o mapas de características con forma 14x14. Antes de pasar a la siguiente capa se pasa por una función de activación **ReLu**.

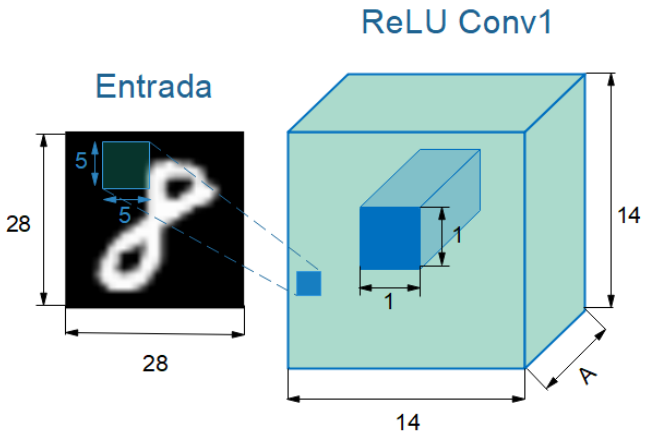

Figura 31. Primera capa de detección de características en Matrix Capsules.

En la segunda etapa de esta red se vuelve a realizar una detección de características. Sin embargo, en esta etapa se cambian los 32 canales anteriores en 32 cápsulas primarias (B=32). Cada una de estas cápsulas contiene una matriz 4x4 y un valor de activación. Por tanto, se usan 4x4+1 neurona para implementar una cápsula. A esta etapa se le llama ***PrimaryCaps*** y usa *kernels* de 1x1x32 (1x1xA) con

un paso de 1 y con relleno. A la salida tendremos 14x14x32 valores de activación y 14x14x32 matrices de pose 4x4.

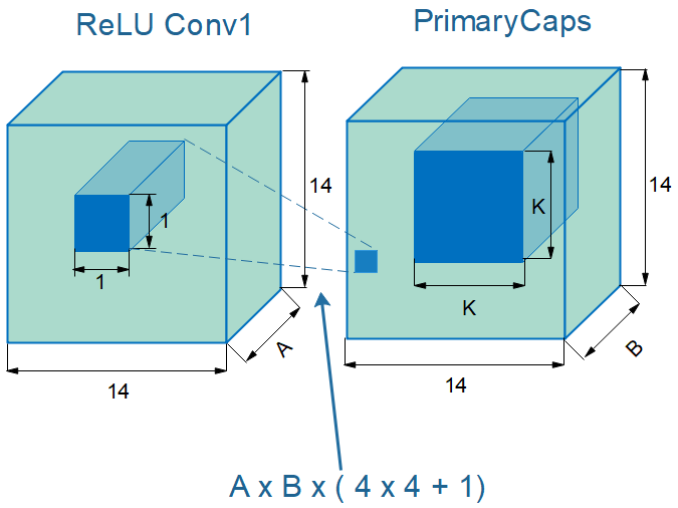

Figura 32. Segunda capa de detección de características y mapeado a cápsulas matriciales.

En la tercera etapa tenemos de nuevo una capa de convolución **ConvCaps1** con filtros 3x3x32 (KxKxB con K=3), con paso 2 y sin relleno. Esta capa toma como entrada 32 cápsulas primarias y da a su salida 32 cápsulas secundarias (C=32). Al tener paso 2 y filtros 3x3 las imágenes con tamaño 14x14 a la entrada ven reducido su tamaña a 6x6 ($\left\lfloor \frac{14-3}{2} + 1 \right\rfloor = 6$). Esta capa es parecida a una capa de convolución normal con la salvedad de que usa el enrutamiento EM para calcular la salida de la cápsula. El enrutamiento EM utiliza la misma **matriz de transformación** W_{ij} a través de la dimensión espacial para calcular los votos. Por tanto, necesitaremos 3x3x32x32x4x4 parámetros para W_{ij} (KxKxBxCx4x4), dado que cada matriz de pose es 4x4. A la salida tendremos 6x6x32 matrices de pose de 4x4 y 6x6x32 valores de activación.

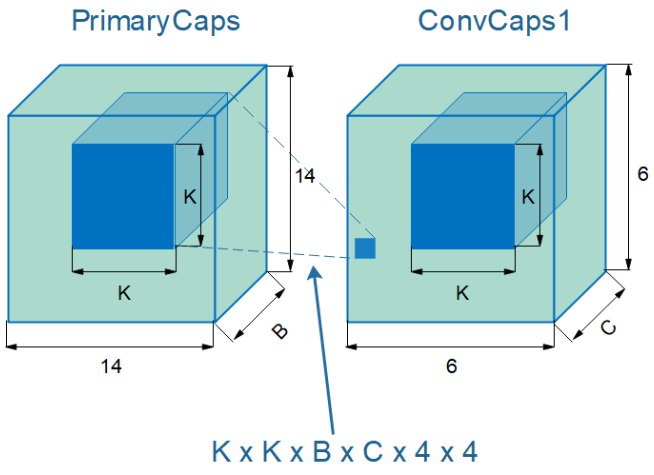

Figura 33. Primera capa de convolución con enrutamiento EM.

La siguiente etapa es una capa similar a la anterior pero con paso 1. Recibe el nombre de **ConvCaps2** y toma como entrada 32 cápsulas y da a su salida 32 cápsulas (D=32). Al tener paso 1 y filtros 3x3x32 **(**KxKxC**)** las imágenes con tamaño 6x6 a la entrada ven reducido su tamaña a 4x4 ($\left\lfloor \frac{6-3}{1} + 1 \right\rfloor = 4$).

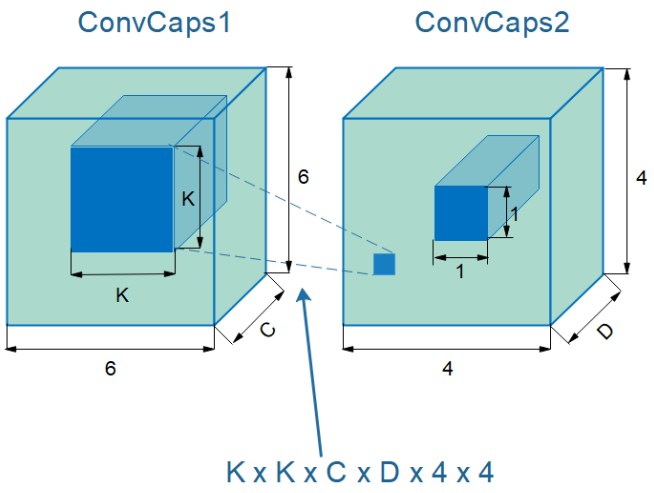

Figura 34. Segunda capa de convolución con enrutamiento EM.

La última etapa está formada por **10 cápsulas de clase** obtenidas empleando un filtro 1x1x32 (1x1xD) con la salida de la capa anterior (E=10). A la salida obtendremos **10 matrices de pose 4x4 y 10 valores de activación**.

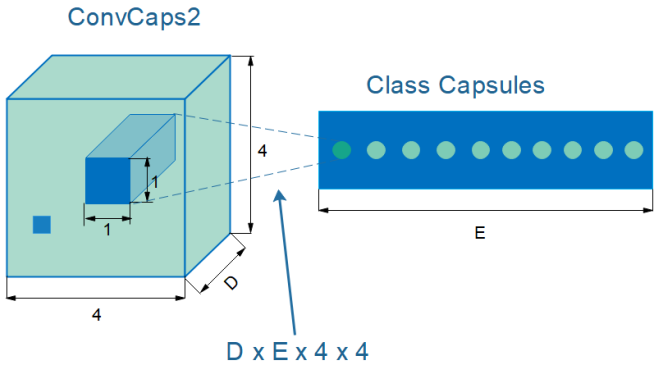

Figura 35. Mapeado con Class Capsules.

Cabe destacar que usamos el **enrutamiento EM** para calcular las matrices de pose y las activaciones de salida para ConvCaps1, ConvCaps2 y Class Capsules.

4.5. Coordinate Addition (Adición de coordenadas)

Al conectar la última capa convolucional a la capa final, no queremos tirar información sobre la ubicación de las cápsulas convolucionales, pero también queremos aprovechar el hecho de que todas las cápsulas del mismo tipo están extrayendo la misma entidad en diferentes posiciones. Por lo tanto, compartimos las matrices de transformación entre las diferentes posiciones del mismo tipo de cápsula y añadimos la coordenada escalada (fila, columna) al centro del campo receptivo de cada cápsula a los primeros 2 elementos empezando por la derecha de su matriz de votos.

Dicho de otra forma, para probar que la red tiene una idea perfecta de la posición **(x,y)** de la entidad a reconocer, por ejemplo una cara, podemos decirle a la red que dibuje dicha cara en las coordenadas **(x+dx, y+dy).** Si la salida coincide con la imagen de la cara desplazada esa cantidad, entonces la red aprendió a almacenar la posición como un par (x,y). En el artículo **(Hinton et al., 2018),** implementaron ***Coordinate Addition*** al agregar la posición del centro del campo receptivo del *kernel* a los primeros dos elementos de la matriz de pose de salida.

Esto debería alentar a las transformaciones finales compartidas a producir valores para esos dos elementos que representan la posición final de la entidad con respecto al centro del campo receptivo de la cápsula.

Al retener la información espacial en la cápsula, nos movemos más allá de simplemente verificar la presencia de una característica. Alentamos al sistema a verificar la relación espacial de las características para evitar adversarios. Es decir, si los órdenes espaciales de las características son incorrectos, sus votos correspondientes no deberían coincidir.

5. Algoritmo de optimización

El proceso mediante el cual la maquina aprende se puede resumir en una idea: la minimización de la función de pérdida mediante la actualización de los pesos de la red. Esto convierte este proceso en un problema de optimización.

En nuestro estudio empleamos el optimizador Adam o ***Adaptative Moment Estimativo***. Este emplea un algoritmo de optimización que permite modificar el valor del rimo de aprendizaje o ***learning rate*** a medida que se desarrolla el entrenamiento. En concreto se trata de una variable del descenso de gradiente estocástico.

La tasa de aprendizaje forma parte de los hiperparámetros de estas redes neuronales. Este parámetro nos permite actualizar los pesos de la red siguiendo la siguiente función:

$$w_j = w_j - \alpha \frac{df(w_j)}{dw_j}$$

Donde w_j es uno de los pesos a ajustar, f es la función de costo o pérdida y α es la tasa de aprendizaje.

Si el valor de esta tasa es demasiado pequeño la convergencia (aprendizaje) de la red llevará demasiado tiempo. Por el contrario, si el valor es demasiado grande, puede ocurrir que el siguiente punto calculado exceda el valor del mínimo haciendo imposible poder alcanzarlo. Por tanto, es necesario escoger un valor de tasa de aprendizaje correcto para el que el funcionamiento de la red sea el deseado.

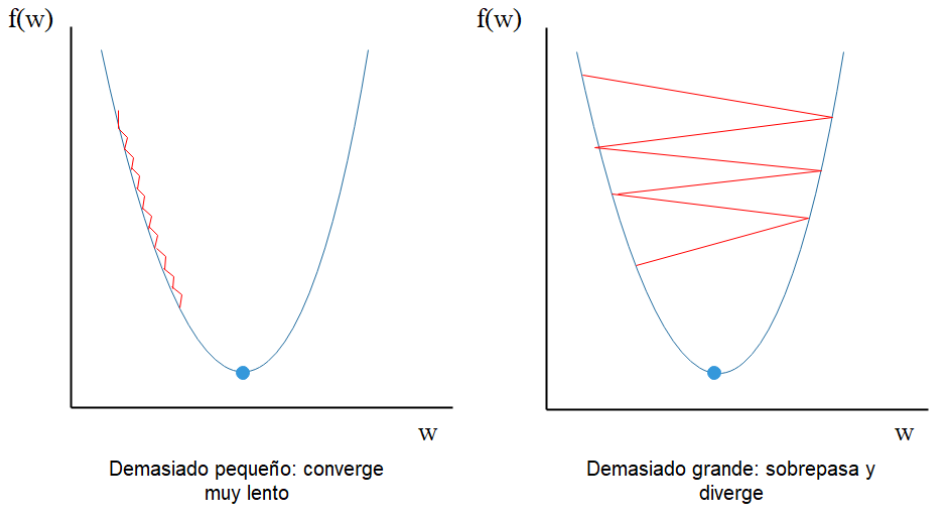

Figura 36. Gráficas de la influencia de la tasa de aprendizaje en la busca de la optimización.

El **proceso de aprendizaje de descenso de gradiente** es un proceso iterativo donde los pesos se actualizan en cada iteración, es por ello que para obtener el mejor resultado es necesario procesar el *dataset* varias veces. Debido a que procesar un *dataset* completo en una sola iteración es complicado por restricciones de memoria este es dividido en partes llamadas ***batch*** de un determinado tamaño. En función del tamaño de este *batch* varia la frecuencia con la que se actualizan los pesos de la red, a menos *batch* mayor frecuencia de actualización.

Además, podemos introducir un término extra al que llamamos ***weigth decay*** (decaimiento de los pesos). Este término de regularización nos permite introducir un valor de corrección extra de los pesos, lo cual nos puede ser útil para solventar problemas de ***overfitting***, explicado en el siguiente apartado.

6. Overfitting o underfitting

Dos de los principales problemas durante el entrenamiento son los llamados ***overfitting* o *underfitting*** de los datos. Si los datos de entrenamiento no son suficientes el sistema no será capaz de reconocer correctamente (***uderfitting***), por lo contrario si se entrena en exceso un modelo este aumentará la precisión sobre el conjunto de entrenamiento pero será incapaz de generalizar y empeorará si recibe datos nuevos (***overfitting***). Podemos ver

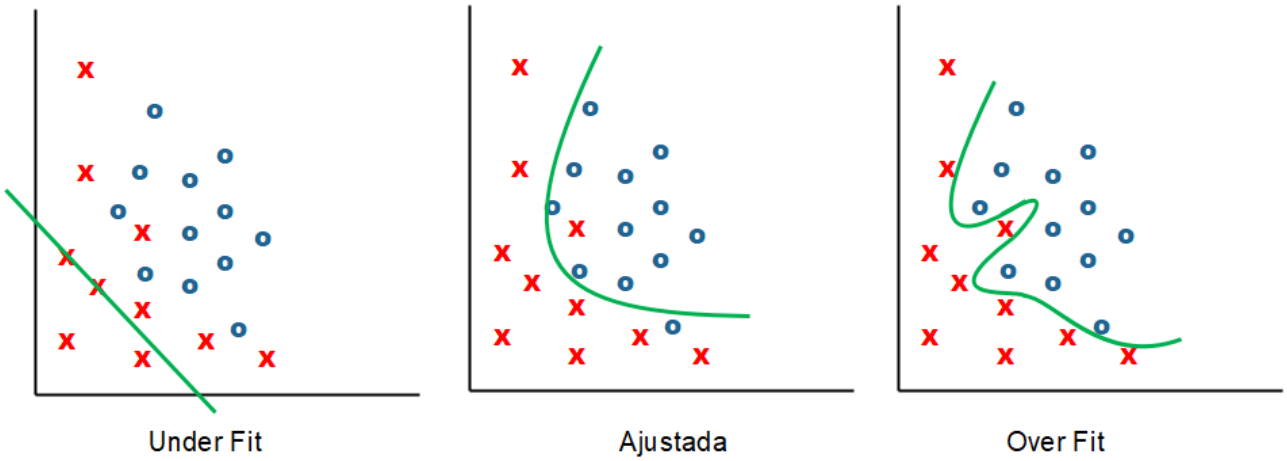

Figura 37. Esquema del ajuste del modelo.

Es por ello que se dividen los datos en dos normalmente, un conjunto de entrenamiento y otro de test, para introducir una gran cantidad de datos distintos a la red con el fin de solventar el ***overfitting*** *y el* ***uderfitting.***

7. Funciones de activación

Una función de activación se encarga de transformar un valor de entrada en un valor de salida con unas características deseadas. Se buscan funciones que las derivadas sean simples, para minimizar con ello el coste computacional.

En lo que respecta a las redes de cápsulas y más concretamente a las implementaciones que hemos escogido para realizar este trabajo podemos hablar de dos funciones de activación en particular. La primera es la función **sigmoide** la cual es la función más antigua y la más popular. Se define como:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Aunque la función parece complicada y arbitraria en realidad tiene una forma bastante simple. La podemos ver en la siguiente imagen:

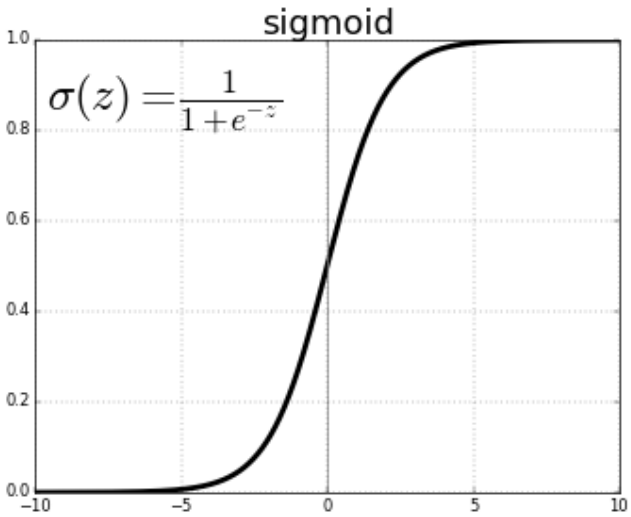

Figura 38. Gráfica de la función sigmoide.

Podemos observar que está acotada entre 0 y 1, lo que puede ser bastante útil en diversos puntos del desarrollo de una red neuronal, como por ejemplo a la hora de querer referenciar un parámetro en forma de probabilidad.

Esta función a pesar de ser la más popular y de haber dominado este terreno durante un largo periodo de tiempo va dejando hueco a otros tipos de funciones de activación, debido a que para redes neuronales de muchas capas comienza a tener ciertos inconvenientes.

Es por ello que entro en escena la función **ReLU** (***rectified linear unit***). Esta función se define como:

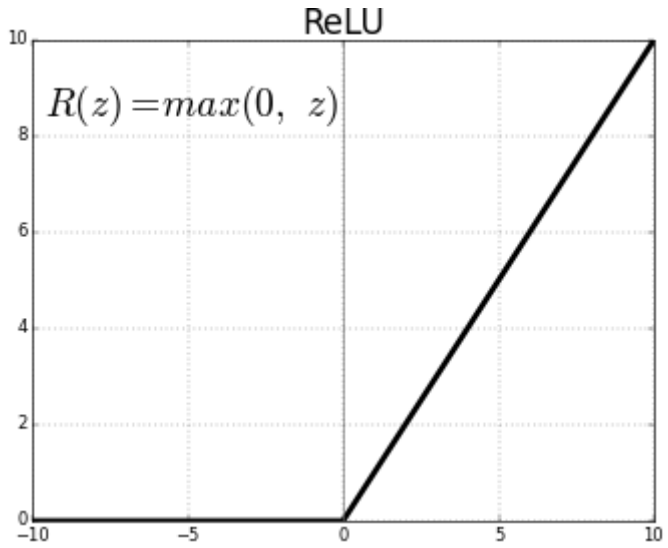

Figura 39. Gráfica de la función ReLU.

Podemos explicar el comportamiento de esta función diciendo que permite el paso de todos los valores positivos sin cambiarlos y asigna a todos los valores negativos el valor de 0.

La mayoría de redes neuronales emplean esta función de activación y cabe destacar su buen desempeño en redes convolucionales.

8. Mediciones

Para realizar la comparación entre los distintos tipos de redes neuronales empleamos la precisión de dicha red. Este valor se calcula dividiendo el número de imágenes clasificadas con éxito entre el número de imágenes total. Este valor puede sufrir variaciones cada vez que ejecutemos la simulación. Es por ello que debemos realizar el promedio y la desviación típica de varios valores de precisión obtenidos tras varias simulaciones.

$$Precisión = \frac{\sum_{i=1}^{N}(Predicción_i == Objetivo_i)}{N}$$

Donde N es el número total de imágenes, $Predicción$ es la etiqueta predicha y $Objetivo$ es la etiqueta objetivo

9. Implementaciones

En este apartado se van a explicar las implementaciones llevadas a cabo con los dos tipos de redes neuronales de cápsulas y para los *datasets* de MNIST, SmallNORB, Cifar10 y retinografías.

9.1. Implementación de CapsNet

Cabe destacar que esta red neuronal ha sido llevada a cabo por Antonio Oliva Aparicio, compañero en la carrera, y usada en su Trabajo de Fin de Grado para compararla con otros tipos de redes.

9.1.1. MNIST

En esta imagen podemos ver la arquitectura de *CapsNet* particularizada para MNIST:

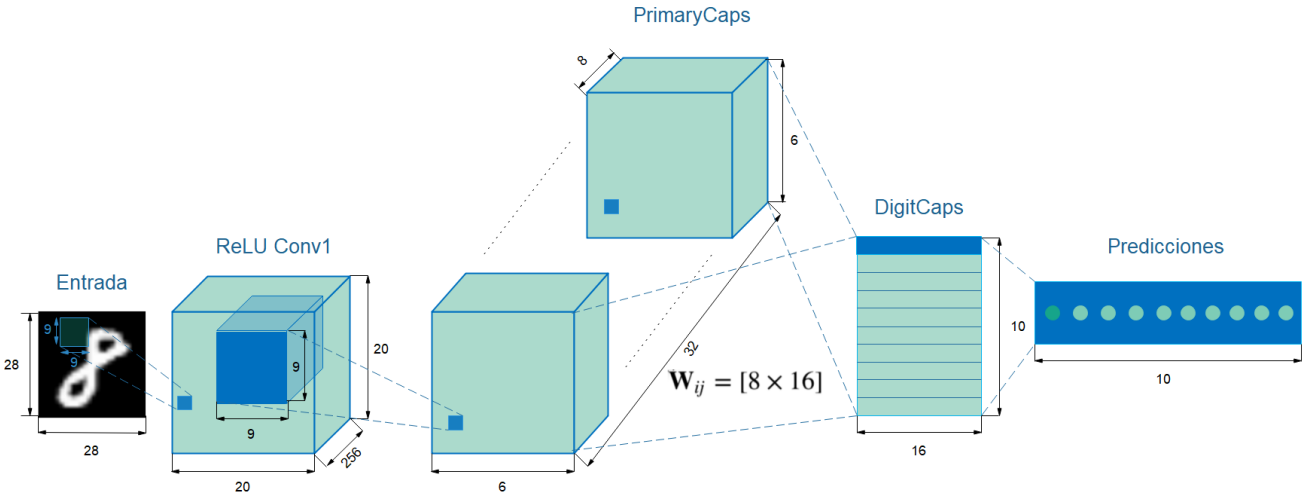

Figura 40. Implementación CapsNet para MNIST.

En este caso seguimos la arquitectura e hiperparámetros que Hinton especifica en su **(Hinton et al., 2018)**. Es decir, una primera capa convolucional (Conv1) con 256 *kernels* de tamaño 9x9, con paso 1 y sin relleno. Un segunda capa convolucional con *kernels* de 9x9x256 con paso 2 y sin relleno que posteriormente (***PrimaryCaps***) divide los 256 mapas de características anteriores en 32 canales de 8 dimensiones. Una tercera capa (***DigitCaps***) que aplica una matriz de transformación W_{ij} de tamaño 16x8 para convertir las 1152 cápsulas anteriores de 8 dimensiones en 10 cápsulas de 16 dimensiones (una por cada digito). Para la reconstrucción se han empleado 3 capas totalmente conectadas con dimensiones de 512, 1024 y 784 respectivamente. La obtención de todos estos valores está mejor explicada en el apartado 10 de este documento.

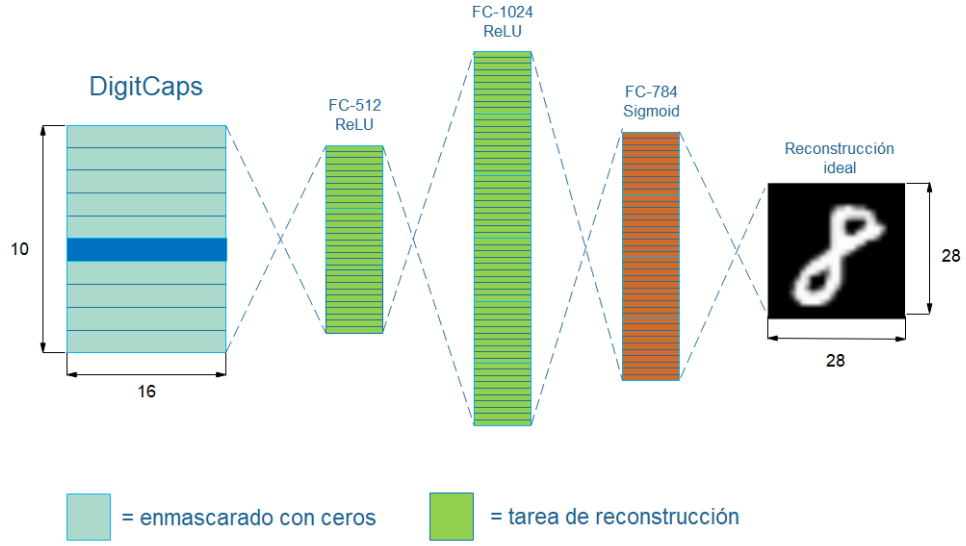

Figura 41. Reconstrucción con CapsNet para MNIST.

9.1.2. SmallNORB

En esta imagen podemos ver la arquitectura de *CapsNet* particularizada para SmallNORB:

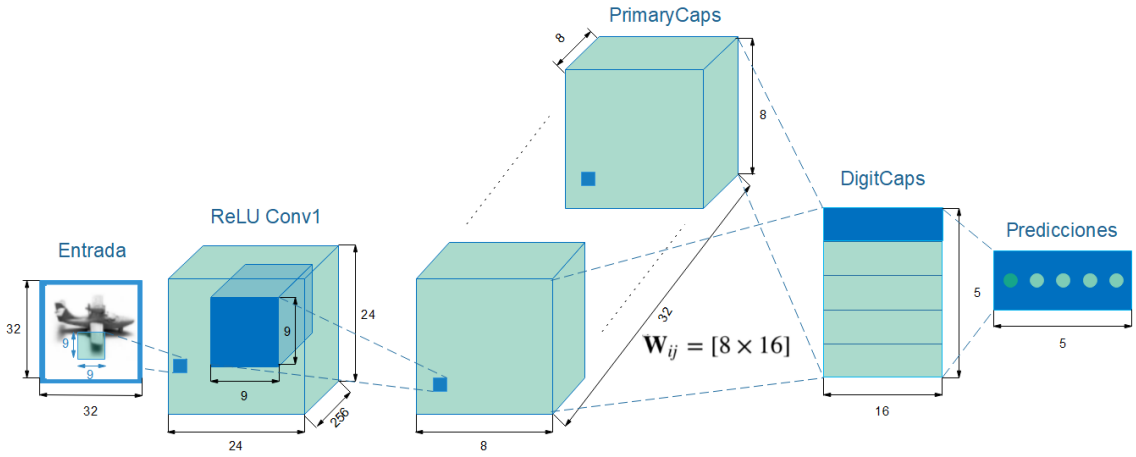

Figura 42. Implementación CapsNet para SmallNORB.

La arquitectura consta de una primera capa convolucional (**Conv1**) que aplica 256 *kernels* de 9x9, con paso 1 sin relleno, lo que reduce el tamaño de la imagen de 32x32 a 24x24 ($\left\lfloor\frac{32-9}{1}\right\rfloor + 1 = 24$) y crea 256 mapas de características. Esta capa tiene una forma de 256x24x24. En la segunda capa (***PrimaryCaps***) se dividen los 256 mapas de características anteriores en 32 canales de cápsulas de 8 dimensiones, usando *kernels* de 9x9x256 con paso 2 sin relleno, lo que reduce el tamaño de la imagen de 24x24 a 8x8 ($\left\lfloor\frac{24-9}{2}\right\rfloor + 1 = 8$). Finalmente, ***PrimaryCaps*** tiene 32x8x8=2048 vectores de longitud 8. La tercera capa (***DigitCaps***) y aplica una matriz de transformación W_{ij} con forma 16x8 para convertir las 2048 cápsulas de 8-D en una cápsula 16-D para cada una de las 5 clases que tiene SmallNORB. Para la reconstrucción se han empleado 3 capas totalmente conectadas con dimensiones de 512, 1024 y 32x32x1=1024.

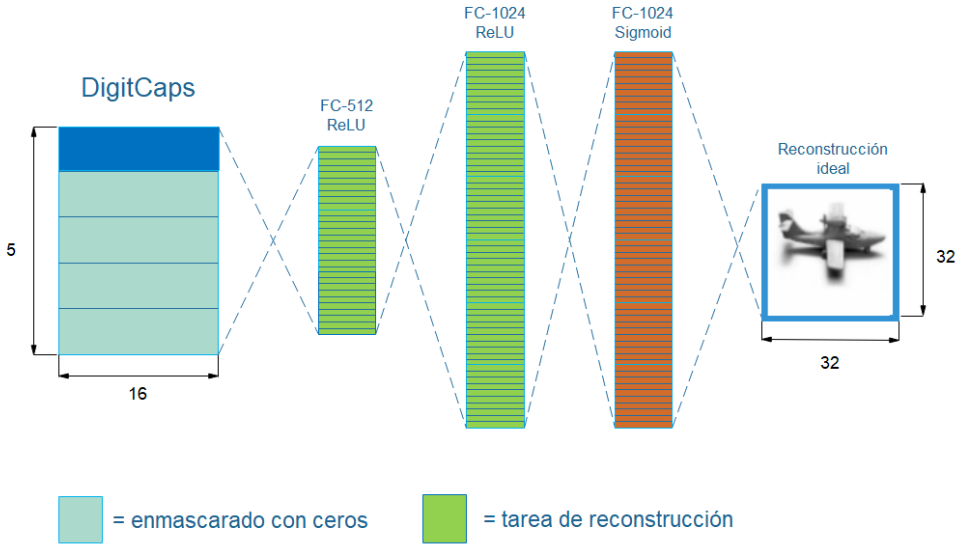

Figura 43. Reconstrucción con CapsNet para SmallNORB.

La imagen de SmallNORB viene con un tamaño de 96x96. Esta imagen ha sido redimensionada a 48x48 y posteriormente recortada de manera aleatoria a 32x32.

9.1.3. Cifar10

En esta imagen podemos ver la arquitectura de *CapsNet* particularizada para Cifar10:

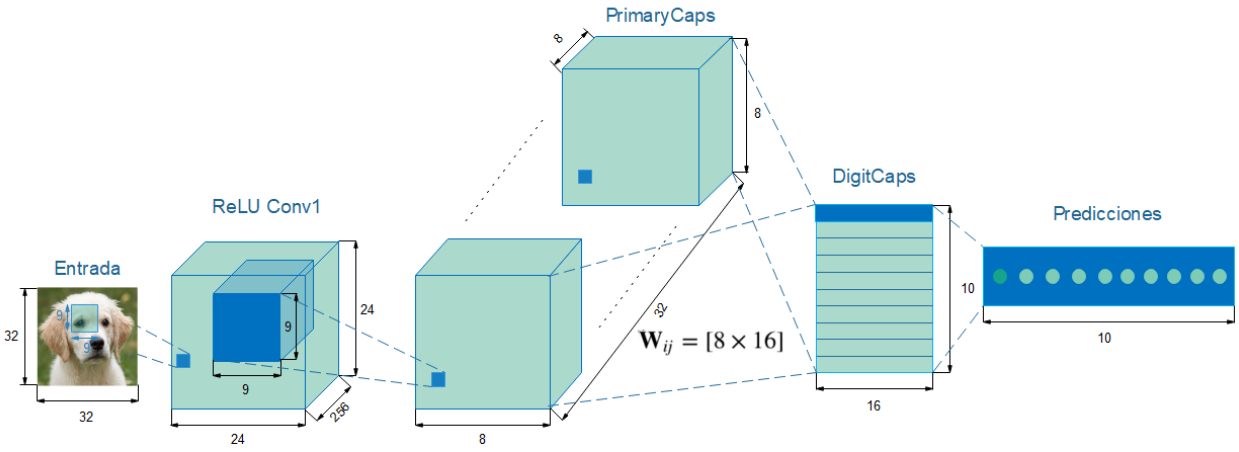

Figura 44. Implementación CapsNet para Cifar10.

La arquitectura de Cifar10 consta de una primera capa convolucional (**Conv1**) que aplica 256 *kernels* de 9x9x3, con paso 1 sin relleno, lo que reduce el tamaño de la imagen de 32x32 a 24x24 ($\left\lfloor \frac{32-9}{1} \right\rfloor + 1 = 24$) y crea 256 mapas de características con tamaño 24x24. Esta capa tiene una forma de 256x24x24. En la segunda capa (***PrimaryCaps***) se dividen los 256 mapas de características anteriores en 32 canales de cápsulas de 8 dimensiones, usando *kernels* de 9x9x256 con paso 2 sin relleno, lo que reduce el tamaño de la imagen de 24x24 a 8x8 ($\left\lfloor \frac{24-9}{2} \right\rfloor + 1 = 8$). Finalmente, ***PrimaryCaps*** tiene 32x8x8=2048 vectores de longitud 8. La tercera capa (***DigitCaps***) y aplica una matriz de transformación W_{ij} con forma 16x8 para convertir las 2048 cápsulas de 8-D en una cápsula 16-D para

cada una de las 5 clases que tiene Cifar10. Para la reconstrucción se han empleado 3 capas totalmente conectadas con dimensiones de 512, 1024 y 32x32x3=3072.

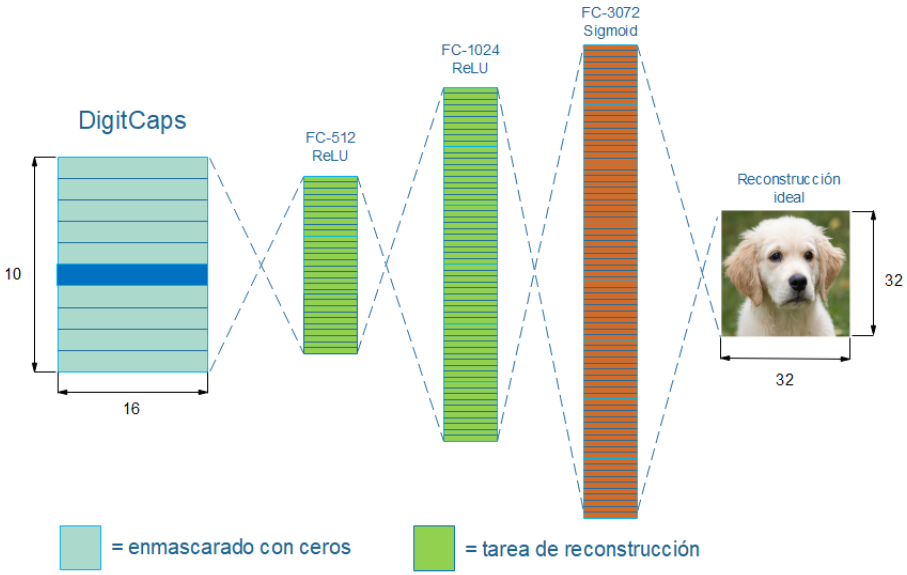

Figura 45. Reconstrucción con CapsNet para Cifar10.

9.1.4. Retinografías

En esta imagen podemos ver la arquitectura de *CapsNet* particularizada para las retinografías:

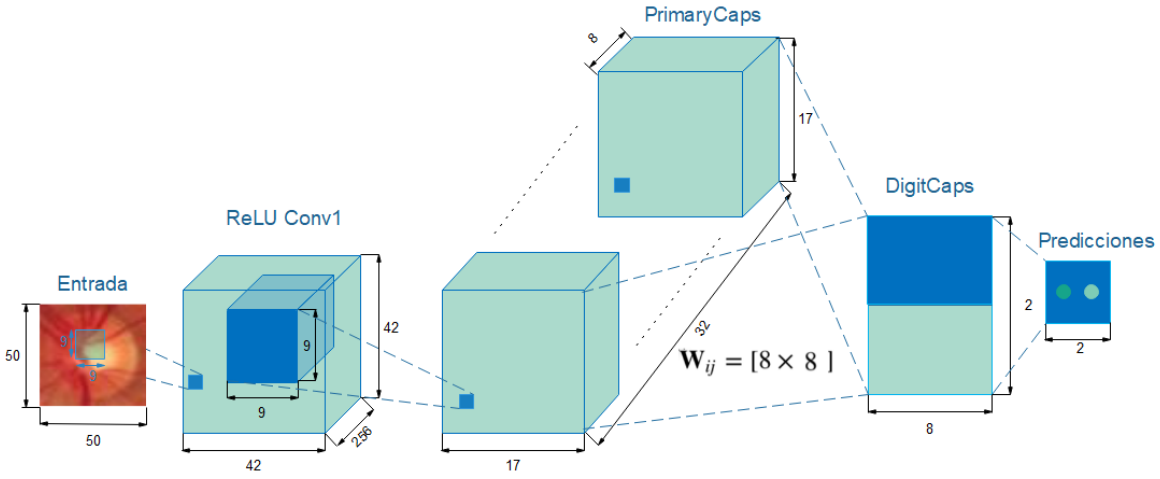

Figura 46. Implementación CapsNet para retinografías.

La arquitectura consta de una primera capa convolucional (**Conv1**) que aplica 256 *kernels* de 9x9x3, con paso 1 sin relleno, lo que reduce el tamaño de la imagen de 50x50 a 42x42 ($\left\lfloor \frac{50-9}{1} \right\rfloor + 1 = 42$) y crea 256 mapas de características. Esta capa tiene una forma de 42x42x256. En la segunda capa (***PrimaryCaps***) se dividen los 256 mapas de características anteriores en 32 canales de cápsulas de 8 dimensiones, usando *kernels* de 9x9x256 con paso 2 sin relleno, lo que reduce el tamaño de la imagen de 42x42 a 17x17 ($\left\lfloor \frac{42-9}{2} \right\rfloor + 1 = 17$). Finalmente, ***PrimaryCaps*** tiene 32x17x17=9248 vectores de longitud 8. La tercera capa (***DigitCaps***) y aplica una matriz de transformación W_{ij} con forma 8x8 para

convertir las 9248 cápsulas de 8-D en una cápsula 8-D para cada una de las 2 clases que tiene el conjunto de retinografías. En este caso para la reconstrucción se han empleado cambiado las dos capas totalmente conectadas intermedias, lo cual no lo habíamos hecho en los anteriores *datasets*, y se han puesto las siguientes dimensiones: 32, 128 y 50x50x3=7500.

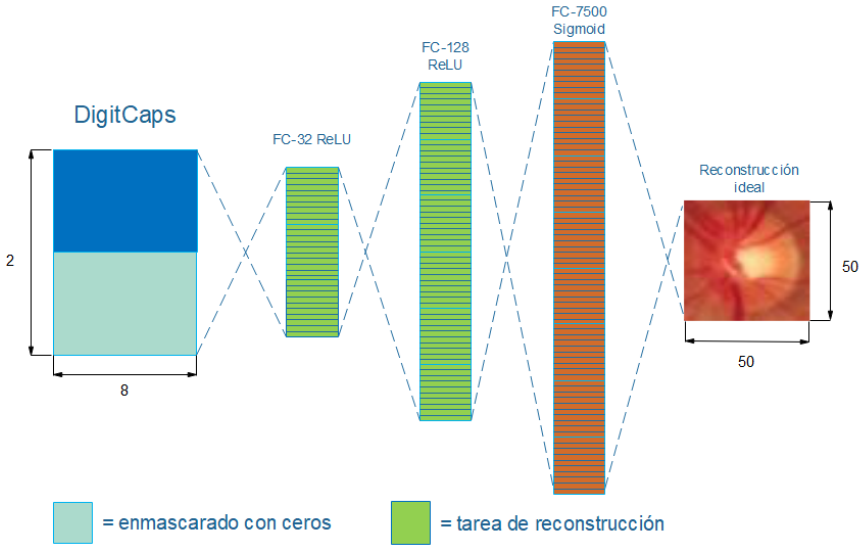

Figura 47. Reconstrucción con CapsNet para retinografías.

9.2. Implementación de Matrix CapsNet

Esta red ha sido desarrollada por **Lei Yang** y la hemos tomado y modificado para adaptarla a los 4 *datasets* que estamos citando en este documento. Cabe destacar que para todos los conjuntos de datos hemos introducido un valor de *learning rate* de 0.01 y 2 iteraciones del algoritmo de EM *routing*. Además, realizamos adicción de coordenadas y una normalización de los datos a la salida de ReLU conv1.

9.2.1. MNIST

En esta imagen podemos ver la arquitectura de *Matrix CapsNet* particularizada para las MNIST:

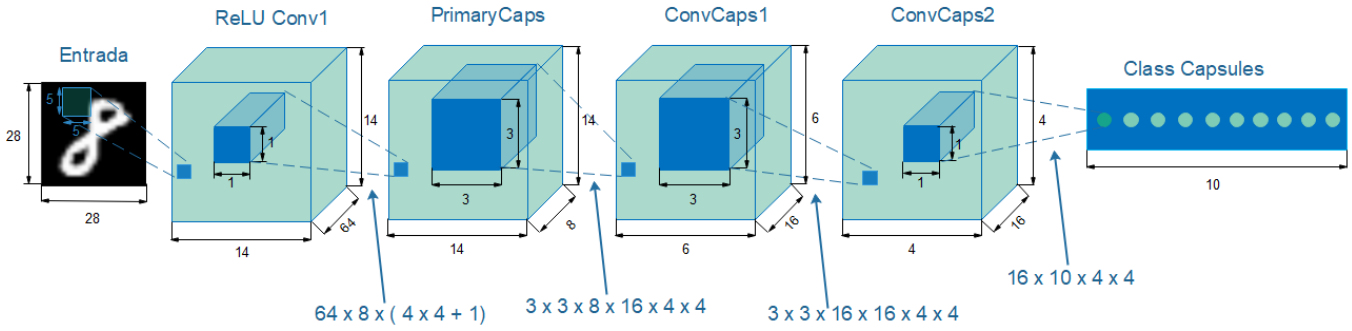

Figura 48. Implementación Matrix CapsNet para MNIST.

Comenzamos con una primera capa de detección de características **Conv1** que aplica 64 *kernels* de 5x5 píxeles a cada imagen de entrada, con un paso de 2 y con relleno, lo cual reduce el tamaño de las imágenes originales de MNIST de 28x28 píxeles a 14x14 píxeles ($\left\lfloor \frac{28+2x2-5}{2} + 1 \right\rfloor = 14$). Se han

elegido 64 filtros para obtener 64 mapas de características. En la segunda capa (***PrimaryCaps***) se pasa de los 64 canales anteriores a 8 cápsulas primarias, cada una de ellas formada por una matriz de pose 4x4 y un valor de activación. Se emplean filtros con *kernels* 1x1x64, con paso 1 con relleno. A la salida tendremos 14x14x8 valores de activación y 14x14x8 matrices de pose 4x4. En la tercera etapa tenemos de nuevo una capa de convolución **ConvCaps1** con filtros 3x3x8, con paso 2 y sin relleno. Esta capa toma como entrada 8 cápsulas primarias y da a su salida 16 cápsulas secundarias. Al tener paso 2 y filtros 3x3 las imágenes con tamaño 14x14 a la entrada ven reducido su tamaña a 6x6 ($\left\lfloor \frac{14-3}{2} + 1 \right\rfloor = 6$). Esta capa es parecida a una capa de convolución normal con la salvedad de que usa el enrutamiento EM para calcular la salida de la cápsula. El enrutamiento EM utiliza la misma matriz de transformación W_{ij} a través de la dimensión espacial para calcular los votos. Por tanto, necesitaremos 3x3x8x16x4x4 parámetros para W_{ij}, dado que cada matriz de pose es 4x4. A la salida tendremos 6x6x16 matrices de pose de 4x4 y 6x6x16 valores de activación. La siguiente etapa es una capa similar a la anterior pero con paso 1. Recibe el nombre de **ConvCaps2** y toma como entrada 16 cápsulas y da a su salida 16 cápsulas. Al tener paso 1 y filtros 3x3x16 las imágenes con tamaño 6x6 a la entrada ven reducido su tamaño a 4x4 ($\left\lfloor \frac{6-3}{1} + 1 \right\rfloor = 4$). La última etapa está formada por 10 cápsulas de clase obtenidas empleando un filtro 1x1x16 con la salida de la capa anterior. A la salida obtendremos 10 matrices de pose 4x4 y 10 valores de activación.

Las imágenes de MNIST han sido normalizadas restándole la media y dividiendo entre su desviación típica para obtener un mejor resultado.

9.2.2. SmallNORB

En esta imagen podemos ver la arquitectura de *Matrix CapsNet* particularizada para las SmallNORB:

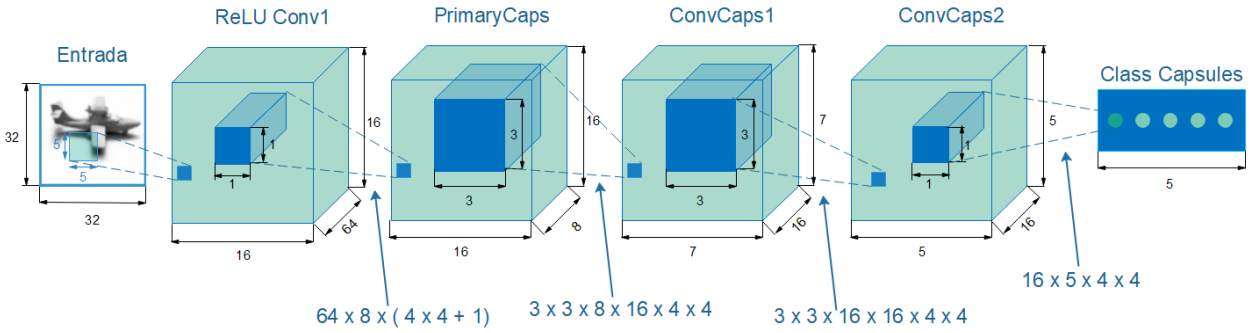

Figura 49. Implementación Matrix CapsNet para SmallNORB.

La primera capa de detección de características **Conv1** aplica 64 *kernel*s de 5x5 píxeles a cada imagen de entrada, con un paso de 2 y con relleno, lo cual reduce el tamaño de las imágenes originales de SmallNORB de 32x32 píxeles a 16x16 píxeles ($\left\lfloor \frac{32+2x2-5}{2} + 1 \right\rfloor = 16$). Se han elegido 64 filtros para obtener 64 mapas de características. En la segunda capa (***PrimaryCaps***) se pasa de los 64 canales anteriores a 8 cápsulas primarias, cada una de ellas formada por una matriz de pose 4x4 y un valor de activación. Se emplean filtros con *kernels* 1x1x64, con paso 1 con relleno. A la salida tendremos 16x16x8 valores de activación y 16x16x8 matrices de pose 4x4. En la tercera etapa tenemos de nuevo una capa de convolución **ConvCaps1** con filtros 3x3x8, con paso 2 y sin relleno. Esta capa toma como entrada 8 cápsulas primarias y da a su salida 16 cápsulas secundarias. Al tener paso 2 y filtros 3x3 las imágenes con tamaño 16x16 a la entrada ven reducido su tamaña a 7x7 ($\left\lfloor \frac{16-3}{2} + 1 \right\rfloor = 7$). Esta capa

es parecida a una capa de convolución normal con la salvedad de que usa el enrutamiento EM para calcular la salida de la cápsula. El enrutamiento EM utiliza la misma matriz de transformación W_{ij} a través de la dimensión espacial para calcular los votos. Por tanto, necesitaremos 3x3x8x16x4x4 parámetros para W_{ij}, dado que cada matriz de pose es 4x4. A la salida tendremos 7x7x16 matrices de pose de 4x4 y 7x7x16 valores de activación. La siguiente etapa es una capa similar a la anterior pero con paso 1. Recibe el nombre de **ConvCaps2** y toma como entrada 16 cápsulas y da a su salida 16 cápsulas. Al tener paso 1 y filtros 3x3x16 las imágenes con tamaño 7x7 a la entrada ven reducido su tamaño a 5x5 ($\left\lfloor \frac{7-3}{1} + 1 \right\rfloor = 5$). La última etapa está formada por 5 cápsulas de clase obtenidas empleando un filtro 1x1x16 con la salida de la capa anterior. A la salida obtendremos 5 matrices de pose 4x4 y 5 valores de activación.

La imagen de SmallNORB viene con un tamaño de 96x96. Esta imagen ha sido redimensionada a 48x48 y posteriormente recortada de manera aleatoria a 32x32.

9.2.3. Cifar10

En esta imagen podemos ver la arquitectura de *Matrix CapsNet* particularizada para las Cifar10:

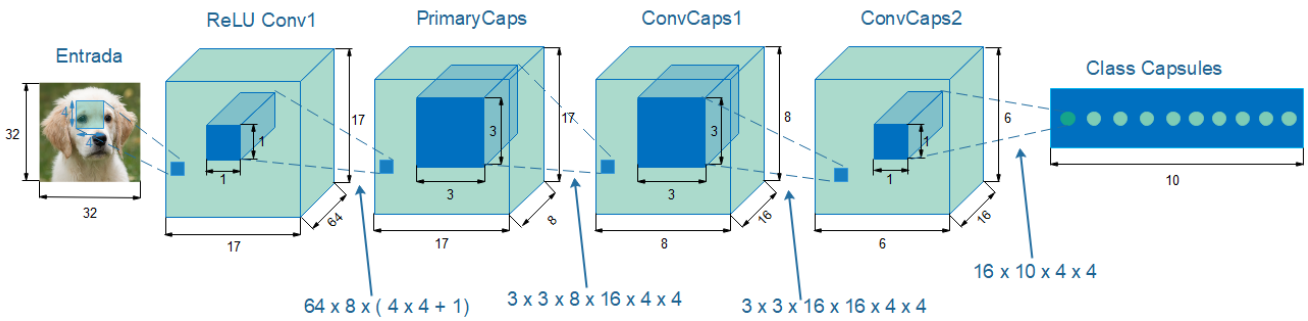

Figura 50. Implementación Matrix CapsNet para Cifar10.

Comenzamos con una primera capa de detección de características **Conv1** que aplica 64 *kernels* de 4x4x3 píxeles a cada imagen de entrada, con un paso de 2 y con relleno, lo cual reduce el tamaño de las imágenes originales de Cifar10 de 32x32 píxeles a 17x17 píxeles ($\left\lfloor \frac{32+2x2-4}{2} + 1 \right\rfloor = 17$). Se han elegido 64 filtros para obtener 64 mapas de características. En la segunda capa (***PrimaryCaps***) se pasa de los 64 canales anteriores a 8 cápsulas primarias, cada una de ellas formada por una matriz de pose 4x4 y un valor de activación. Se emplean filtros con *kernels* 1x1x64, con paso 1 con relleno. A la salida tendremos 17x17x8 valores de activación y 17x17x8 matrices de pose 4x4. En la tercera etapa tenemos de nuevo una capa de convolución **ConvCaps1** con filtros 3x3x8, con paso 2 y sin relleno. Esta capa toma como entrada 8 cápsulas primarias y da a su salida 16 cápsulas secundarias. Al tener paso 2 y filtros 3x3 las imágenes con tamaño 17x17 a la entrada ven reducido su tamaño a 8x8 ($\left\lfloor \frac{17-3}{2} + 1 \right\rfloor = 8$). Esta capa es parecida a una capa de convolución normal con la salvedad de que usa el enrutamiento EM para calcular la salida de la cápsula. El enrutamiento EM utiliza la misma matriz de transformación W_{ij} a través de la dimensión espacial para calcular los votos. Por tanto, necesitaremos 3x3x8x16x4x4 parámetros para W_{ij}, dado que cada matriz de pose es 4x4. A la salida tendremos 8x8x16 matrices de pose de 4x4 y 8x8x16 valores de activación. La siguiente etapa es una capa similar a la anterior pero con paso 1. Recibe el nombre de **ConvCaps2** y toma como entrada 16

cápsulas y da a su salida 16 cápsulas. Al tener paso 1 y filtros 3x3x16 las imágenes con tamaño 8x8 a la entrada ven reducido su tamaño a 6x6 ($\left\lfloor \frac{8-3}{1} + 1 \right\rfloor = 6$). La última etapa está formada por 10 cápsulas de clase obtenidas empleando un filtro 1x1x16 con la salida de la capa anterior. A la salida obtendremos 10 matrices de pose 4x4 y 10 valores de activación.

9.2.4. Retinografías

En esta imagen podemos ver la arquitectura de *Matrix CapsNet* particularizada para las retinografías:

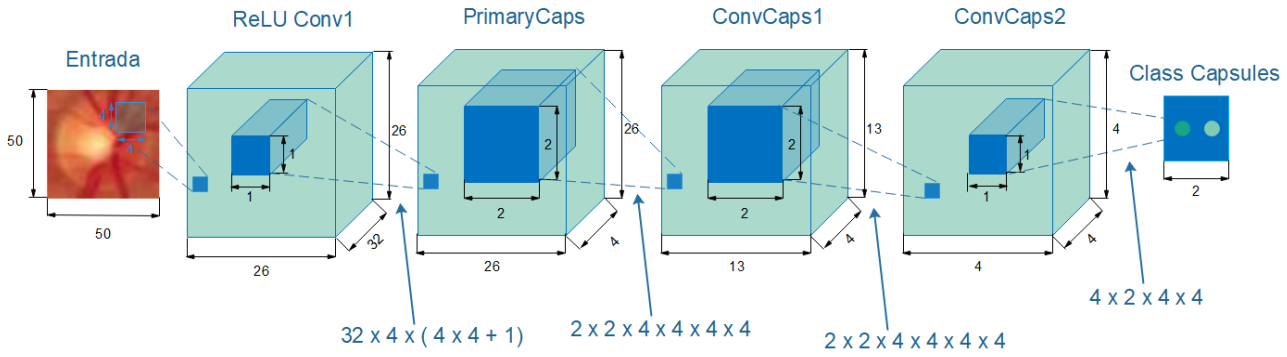

Figura 51. Implementación Matrix CapsNet para retinografías.

La primera capa de detección de características **Conv1** aplica 32 *kernels* de 4x4x3 píxeles a cada imagen de entrada, con un paso de 2 y con relleno, lo cual reduce el tamaño de las retinografías originales de 50x50 píxeles a 26x26 píxeles ($\left\lfloor \frac{50+2x2-4}{2} + 1 \right\rfloor = 26$). Se han elegido 32 filtros para obtener 32 mapas de características. En la segunda capa (***PrimaryCaps***) se pasa de los 32 canales anteriores a 4 cápsulas primarias, cada una de ellas formada por una matriz de pose 4x4 y un valor de activación. Se emplean filtros con *kernels* 1x1x32, con paso 1 con relleno. A la salida tendremos 26x26x4 valores de activación y 26x26x4 matrices de pose 4x4. En la tercera etapa tenemos de nuevo una capa de convolución **ConvCaps1** con filtros 2x2x4 (en lugar de 3x3x4 como en los *datasets* anteriores), con paso 2 y sin relleno. Esta capa toma como entrada 4 cápsulas primarias y da a su salida 4 cápsulas secundarias. Al tener paso 2 y filtros 2x2 las imágenes con tamaño 26x26 a la entrada ven reducido su tamaño a 12x12 ($\left\lfloor \frac{26-2}{2} + 1 \right\rfloor = 13$). Esta capa es parecida a una capa de convolución normal con la salvedad de que usa el enrutamiento EM para calcular la salida de la cápsula. El enrutamiento EM utiliza la misma matriz de transformación W_{ij} a través de la dimensión espacial para calcular los votos. Por tanto, necesitaremos 2x2x8x16x4x4 parámetros para W_{ij}, dado que cada matriz de pose es 4x4. A la salida tendremos 13x13x16 matrices de pose de 4x4 y 13x13x16 valores de activación. La siguiente etapa es una capa similar a la anterior pero con paso 1. Recibe el nombre de **ConvCaps2** y toma como entrada 4 cápsulas y da a su salida 4 cápsulas. Al tener paso 1 y filtros 2x2x4 las imágenes con tamaño 13x13 a la entrada ven reducido su tamaño a 12x12 ($\left\lfloor \frac{13-2}{1} + 1 \right\rfloor = 12$). La última etapa está formada por 2 cápsulas de clase obtenidas empleando un filtro 1x1x4 con la salida de la capa anterior. A la salida obtendremos 2 matrices de pose 4x4 y 2 valores de activación.

10. Capas y número de parámetros

En este apartado, vamos a realizar un estudio del número de parámetros que han sido necesarios para llevar a cabo la comparación de dos de las implementaciones explicadas en este documento: *CapsNet* y *Matrix CapsNet*.

10.1. CapsNet

La arquitectura de esta red ha sido explicada anteriormente en el apartado 9 y en la siguiente tabla podemos ver la estructura particularizada para cada uno de los *datasets* empleados:

	Conv1	Conv2	Canales	PrimaryCaps	DigitCaps	Decodificador
MNIST	256 de 9x9x1	256 de 9x9x256	32	8 dimensiones	16 dimensiones	512-1024-784-10
SmallNORB	256 de 9x9x1	256 de 9x9x256	32	8 dimensiones	16 dimensiones	512-1024-1024-10
Cifar10	256 de 9x9x3	256 de 9x9x256	32	8 dimensiones	16 dimensiones	512-1024-3072-10
Retinografías	256 de 9x9x3	256 de 9x9x256	32	8 dimensiones	8 dimensiones	32-128-7500-10

Teniendo en cuenta la tabla anterior y el tamaño de las entradas a cada capa podemos obtener el número de parámetros de entrenamiento:

	Conv1	Conv2	Enrutamiento	Total
MNIST	256x(9x9x1+1)	256x(9x9x256+1)	10x(1152x8x16+1152x2)	6,827,264
SmallNORB	256x(9x9x1+1)	256x(9x9x256+1)	5x(2048x8x16+2048x2)	6,660,864
Cifar10	256x(9x9x3+1)	256x(9x9x256+1)	10x(2048x8x16+2048x2)	8,033,536
Retinografías	256x(9x9x3+1)	256x(9x9x256+1)	2x(2500x8x8+2500x2)	5,701,136

	Antes de la reconstrucción	Decodificador	Total
MNIST	6,827,264	512x(16x10+1)+1024x(512+1)+(28x28x1)x(1024+1)	8,238,608
SmallNORB	6,660,864	512x(16x5+1)+1024x(512+1)+(32x32x1)x(1024+1)	8,277,248
Cifar10	8,033,536	512x(16x10+1)+1024x(512+1)+(32x32x3)x(1024+1)	11,790,080
Retinografías	5,701,136	32x(8x2+1)+128x(32+1)+(50x50x3)x(128+1)	6,673,404

A continuación, vamos a explicar el cálculo de los parámetros anteriores particularizando en el caso de MNIST pero pudiéndose extender al resto de *datasets*.

En una capa convolucional, el número de parámetros se calcula multiplicando el número de núcleos por tamaño de dichos núcleos y sumando uno 1 de sesgo.

A la salida de las capas convolucionales se produce una reducción de tamaño de las imágenes de entrada. Cuando no hay relleno, la reducción se calcula restando a la dimensión de la imagen la dimensión del núcleo, dividiendo entre el paso y sumando uno de sesgo. Por ejemplo, a la salida de la capa convolucional Conv1, la cual tiene un paso de 1, no tiene relleno y usa núcleos 9x9, obtenemos una imagen de salida de tamaño ($\left\lfloor \frac{28-9}{1} + 1 \right\rfloor = 20$). Cuando hay relleno, la reducción se calcula con la siguiente fórmula:

$$Salida = \left\lfloor \frac{Entrada + 2xRelleno - nucleo}{paso} + 1 \right\rfloor$$

El número de cápsulas primarias se calcula multiplicando el número de canales por el tamaño de la imagen. Por tanto, en el caso de MNIST tenemos 6x6x32=1152 cápsulas primarias. Es decir, necesitamos 1152 matrices de transformación W_{ij} de dimensión 8x16 y 1152 coeficientes c_{ij} y b_{ij} para cada cápsula primaria y 10 cápsulas primarias (una para cada digito). Resumiendo, necesitamos 10x(1152x8x16+1152x2) parámetros para el enrutamiento.

Además, a todo esto se le suma el número de parámetros de la capa de reconstrucción. Esta capa está compuesta por capas completamente conectadas. Para calcular el número de parámetros necesarios para este tipo de cápsulas. En capas completamente conectadas, se multiplica el número de salidas por el número de entradas más uno, debido al sesgo. La entrada tiene dimensión de una cápsula de digito por el número de cápsulas de digito que hay. La salida es un vector plano cuya longitud es el tamaño total de los datos de entrada. Para el caso de MNIST, 28x28x1 = 784.

10.2. Matrix CapsNet

La arquitectura de esta red también ha sido explicada en el apartado 9 y en la siguiente tabla podemos ver la red particularizada para los distintos conjuntos de datos:

	ReLu Conv1	PrimaryCaps	ConvCaps1	ConvCaps2	Class Capsules
MNIST	A=64 filtros de 5x5x1	B=8 canales	C=16 filtros de 3x3	D=16 filtros de 3x3	E=10
SmallNORB	A=64 filtros de 5x5x1	B=8 canales	C=16 filtros de 3x3	D=16 filtros de 3x3	E=5
Cifar10	A=64 filtros de 4x4x3	B=8 canales	C=16 filtros de 3x3	D=16 filtros de 3x3	E=10
Retinografías	A=32 filtros de 4x4x3	B=4 canales	C=4 canales filtros de 2x2	D=4 canales filtros de 2x2	E=2

Teniendo en cuenta la tabla anterior y el tamaño de las entradas a cada capa podemos obtener el número de parámetros de entrenamiento:

	ReLu Conv1	PrimaryCaps	ConvCaps1	Salida de ConvCaps1
MNIST	64x(5x5x1+1)	64x8x(4x4+1)	3x3x8x16x4x4	28,800
SmallNORB	64x(5x5x1+1)	64x8x(4x4+1)	3x3x8x16x4x4	28,800
Cifar10	64x(4x4x3+1)	64x8x(4x4+1)	3x3x8x16x4x4	30,272
Retinografías	32x(4x4x3+1)	32x4x(4x4+1)	2x2x4x4x4x4	4,768

	Salida de ConvCaps1	ConvCaps2	Class Capsules	Total
MNIST	28,800	3x3x16x16x4x4	16x10x4x4	68,224
SmallNORB	28,800	3x3x16x16x4x4	16x5x4x4	66,944
Cifar10	30,272	3x3x16x16x4x4	16x10x4x4	69,696
Retinografías	4,768	2x2x4x4x4x4	4x2x4x4	5,920

Podemos ver que el número de parámetros de una red de cápsulas matricial es mucho menor que el una red de cápsulas vectorial, como se nos especificaba en **(Hinton et al., 2018)**. Lo cual es una gran ventaja porque necesitaremos una menor cantidad de memoria para almacenarlos.

Ahora, vamos a explicar el cálculo de estos parámetros y al igual que hemos hecho con CapsNet tomaremos como ejemplo MNIST pero se puede extender para el resto de conjuntos de datos:

En la primera capa ReLu Conv1, la forma de calcular el número de parámetros es la misma que la explicada en el apartado anterior para capas convolucionales.

El número de parámetros en las cápsulas primarias se calcula multiplicando el número de canales de entrada por el número de canales a la salida por el tamaño de la matriz de pose más un valor de activación.

En el caso de la capa ConvCaps1 necesitaremos una matriz de transformación W_{ij} que será usada en el enrutamiento para calcular los votos. Para calcular esta matriz necesitaremos (KxKxBxCx4x4) siendo KxK el tamaño del filtro de la convolución, 4x4 el tamaño de la matriz de pose y B y C las cápsulas de entrada y salida de la capa.

La capa ConvCaps1 es similar a la anterior y podemos calcular de igual manera el número de parámetros.

La última capa requerirá una matriz 4x4 por cada clase de cápsula E (una por cada tipo de imagen que tiene el *dataset*) y por cada cápsula de salida de la capa anterior ConvCaps2 (DxEx4x4)

11. Resultados

En este apartado vamos a comparar los valores de precisión obtenidos en las implementaciones del apartado 9. Para ello, hemos puesto en la siguiente tabla la media y la desviación típica (std) de la precisión de las simulaciones realizadas:

	CapsNet		Matrix CapsNet	
	Media (%)	Std (%)	Media (%)	Std (%)
MNIST	99.31	0.01	99.07	0.04
SmallNORB	88.69	0.16	87.67	0.53
Cifar10	72.42	0.21	66.93	0.23
Retinografías	94.08	2.60	96.7	2.24

En el caso de CapsNet se han empleado *batch size* de tamaño de 32 muestras para MNIST y Cifar10, de tamaño de 10 muestras para SmallNORB y de 128 para retinografías. A su vez se han empleado 30 épocas para MNIST, 35 para SmallNORB, 50 para Cifar10 y 300 para retinografías. Las simulaciones de *CapsNet* han sido realizadas con un ordenador portátil con una CPU con unas prestaciones sencillas.

Por otra parte, para *Matrix CapsNet* se han empleado *batch size* de tamaño 32 muestras para MNIST y SmallNORB y de tamaño 20 para retinografías y Cifar10. Para MNIST se han utilizado 40 épocas, 35 para Cifar10, 25 para SmallNORB y 300 para retinografías. Estas simulaciones se han realizado mediante el software proporcionado por Google llamado Google Colab. Este nos ha facilitado el uso de GPUs con mejores prestaciones que una CPU normal. Sin embargo, este software nos ha puesto otros impedimentos. Uno de ellos es que el tiempo máximo que puede durar una sesión de Google Colab es de 12 horas. Por tanto, esto nos limitaba el número de épocas que podíamos estar ejecutando las simulaciones. Por otra parte, las GPUs tienen una memoria limitada. Esto nos limitaba el tamaño de las redes neuronales que podíamos introducir. Estos dos inconvenientes limitan las prestaciones de nuestras implementaciones.

En el conjunto de SmallNORB podemos ver un posible problema de *overfitting* puesto que la precisión del conjunto de pruebas es inferior a la precisión del conjunto de entrenamiento. Hemos intentando solventar esto de diversas maneras entre ellas cambiando el tamaño de la red, disminuyendo el batch size, introduciendo una normalización extra del modelo mediante un *weigth decay y* tratando la imagen antes de introducirla al modelo (restando la media y dividiendo entre la varianza, alterando el brillo y contraste o cambiando su dimensión). Estos cambios no surgieron el efecto deseado y no conseguimos arreglar el problema.

A continuación, vamos a comparar los resultados obtenidos con nuestras simulaciones. Por una parte podemos ver que los resultados son muy semejantes. Hemos obtenido una precisión mejor en los datasets de Cifar10, MNIST y SmallNORB con CapsNet. Sin embargo, hemos encontrado mejora en retinografías con Matrix CapsNet.

Cabe destacar que debido a que no hemos usado los mismas técnicas que han empleado los desarrolladores de los documentos de **(Hinton et al., 2018)** y **(Sabour et al., 2017)** y no hemos conseguido obtener tanta precisión como ellos. Consiguen precisiones superiores al 97% con SmallNORB, 99.5% con MNIST y 88% Cifar10, empleando técnicas como *data augmentation*, explicada en el anexo B, o una técnica de ensamblaje de 7 modelos, probando hasta 64 tipos de *PrimaryCaps* entre otras.

12. Conclusiones

En este documento hemos podido apreciar la importancia de la elección de los hiperparámetros de las redes neuronales. Además, el correcto procesado de los datos de entrada también es un factor importante a tener en cuenta. Estos dos factores definirán la precisión y el tiempo de ejecución del modelo. En nuestro caso podemos recalcar que, aunque como hemos dicho no hemos obtenido resultados tan buenos como en los *papers* oficiales de los creadores, hemos conseguido unos resultados relativamente próximos a los resultados con una tecnología de redes neuronales en comienzos de desarrollo.

Además, hemos conseguido una precisión superior al 94% con el *dataset* de las retinografías. Estos resultados son muy buenos en un conjunto de datos que puede tener mucha más relevancia socialmente, sobretodo en el área de la sanidad, que cualquiera de los otros *dataset* que hemos tratado en este documento.

Cabe destacar también como hemos visto que el número de parámetros necesarios en una red neuronal matricial es mucho menor que el de cualquier otra red existente. Esto nos permite reducir la cantidad de memoria necesaria para llevar a cabo nuestras simulaciones.

Para quien desee continuar con estas implementaciones e intente mejorarlas le aconsejaría que usase técnicas de altas prestaciones, como la ampliación del conjunto de datos de entrada, primero para intentar solventar el problema del sobreajuste de SmallNORB y seguidamente para aumentar la precisión del resto de conjuntos de datos de entrada.

Anexos

A. Dataset (Conjunto de datos)

En este anexo explicaremos los distintos tipos de *dataset* que han sido empleados para realizar este trabajo. Como hemos nombrado, son 4: MNIST, SmallNorb, Cifar10 y retinografías.

A.1. MNIST

La base de datos de MNIST consta de 60,000 ejemplos en su conjunto de entrenamiento y 10,000 ejemplos en su conjunto de prueba. Estos ejemplos son dígitos escritos a mano y forman un subconjunto de un conjunto más grande llamado NIST. Estos dígitos han sido normalizados por tamaño y centrados en una imagen de tamaño fijo.

Esta normalización se ha llevado a cabo con el fin de que las imágenes originales en blanco y negro de NIST cogiesen en un cuadro de 20x20 píxeles, conservando su relación en aspecto. Las imágenes resultantes contienen niveles de gris como resultado de la normalización y están centradas en una imagen de 28x28 calculando el centro de masa de los píxeles y colocando dicho punto en el centro del campo de 28x28.

La base de datos MNIST se construyó a partir de la Base de datos NIST's *Special Database* 3 y la Base de datos NIST's *Special Database* 1 de NIST que contienen imágenes binarias de dígitos escritos a mano. El NIST originalmente designó SD-3 como su conjunto de entrenamiento y SD-1 como su conjunto de prueba. Sin embargo, SD-3 es mucho más limpio y fácil de reconocer que SD-1. La razón de esto se puede encontrar en el hecho de que el SD-3 fue recolectado entre los empleados de la Oficina del Censo, mientras que el SD-1 fue recolectado entre los estudiantes de secundaria. Sacar conclusiones sensatas de los experimentos de aprendizaje requiere que el resultado sea independiente de la elección del conjunto de entrenamiento y la prueba entre el conjunto completo de muestras. Por lo tanto, era necesario construir una nueva base de datos mezclando los conjuntos de datos de NIST.

De esta manera se creó MNIST que consta de 30,000 patrones de SD-3 y 30,000 patrones de SD-1 como conjunto de entrenamiento y de 5,000 patrones de SD-3 y 5,000 patrones de SD-1 como conjunto de pruebas. El conjunto de entrenamiento contiene datos de aproximadamente de 250 escritores, asegurándonos de que el conjunto de escritores de ambos conjuntos de datos sean disjuntos.

SD-1 contiene 58,527 imágenes de dígitos escritas por 500 escritores diferentes. A diferencia de SD-3, donde los bloques de datos de cada escritor aparecieron en secuencia, los datos en SD-1 están mezclados. Las identidades de escritor para SD-1 están disponibles y utilizamos esta información para descifrar los escritores. Luego dividimos SD-1 en dos: los caracteres escritos por los primeros 250 escritores entraron en nuestro nuevo conjunto de entrenamiento. Los 250 escritores restantes se colocaron en nuestro conjunto de prueba. Por lo tanto, tuvimos dos conjuntos con casi 30,000 ejemplos cada uno. El nuevo conjunto de entrenamiento se completó con suficientes ejemplos de SD-3, comenzando en el patrón número 0, para hacer un conjunto completo de 60,000 patrones de entrenamiento. Del mismo modo, el nuevo conjunto de prueba se completó con ejemplos SD-3 comenzando en el patrón número 35,000 para hacer un conjunto completo con 60,000 patrones de prueba, de las cuales solo usaremos un subconjunto de 10,000 imágenes (5,000 de SD-1 y 5, 000 de SD-3).

Los datos se almacenan en un formato de archivo muy simple diseñado para almacenar vectores y matrices multidimensionales. Este formato se llama IDX.

Hay 4 archivos:

train-images-idx3-ubyte:imágenes del conjunto de entrenamiento.
t10k-images-idx3-ubyte: imágenes del conjunto de prueba.

Estos dos archivos siguen la estructura mostrada en la tabla:

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
……			
xxxx	unsigned byte	??	pixel

Los pixeles toman valores entre 0 y 255, siendo 0 blanco y 255 negro.

train-labels-idx1-ubyte: etiquetas del conjunto de entrenamiento
t10k-labels-idx1-ubyte: etiquetas del conjunto de prueba

Estos dos archivos siguen la estructura mostrada en la tabla:

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
……			
xxxx	unsigned byte	??	label

La etiqueta o label toma valores entre 0 y 9

Magic Number es un número entero *MSB first* (*most significant bit first*) el cual consta de 4 bytes. Los 2 primeros bytes son siempre 0. El tercer byte codifica el tipo de datos siendo:

0x08: byte sin signo
0x09: byte con signo
0x0B: short (2 bytes)
0x0C: int (4 bytes)
0x0D: flotante (4 bytes)
0x0E: doble (8 bytes)

El cuarto byte codifica el número de dimensiones del vector/matriz. Los tamaños en cada dimensión son enteros de 4 bytes *MSB first*.

En la siguiente imagen podemos ver diversas muestras del conjunto de MNIST:

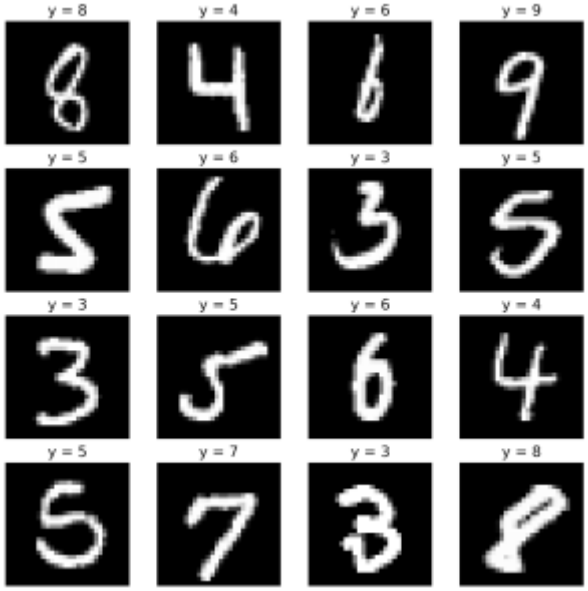

Figura 52. Muestras de MNIST.

A.2. SmallNORB

Esta base de datos es comúnmente usada en experimentos de reconocimiento de objetos 3D a partir de formas. Contiene imágenes de 50 juguetes pertenecientes a 5 categorías genéricas: animales de 4 patas, figuras humanos, aviones, camiones y automóviles. Estos objetos fueron fotografiados por dos cámaras bajo 6 condiciones de iluminación, 9 elevaciones (30 a 70 grados cada 5 grados) y 18 acimuts (0 a 340 cada 20 grados).

El conjunto de entrenamiento se compone de 5 instancias de cada categoría (instancias 4, 6, 7, 8 y 9), y el conjunto de prueba de las 5 instancias restantes (instancias 0, 1, 2, 3 y 5).

Los archivos están en un formato de matriz binaria simple, con el archivo postfix ".mat".

Los archivos "-dat" almacenan las secuencias de imágenes. Los archivos "-cat" almacenan la categoría correspondiente de las imágenes. Cada archivo "-dat" almacena 24,300 pares de imágenes (5 categorías, 5 instancias, 6 iluminaciones, 9 elevaciones y 18 acimuts). El archivo "-cat" correspondiente contiene 24,300 etiquetas de categoría (0 para animales, 1 para humanos, 2 para avión, 3 para camión, 4 para automóvil).

Cada archivo "-info" almacena 24,300 vectores de 4 dimensiones, que contienen información adicional sobre las imágenes correspondientes:

1. La instancia en la categoría (0 a 9).
2. La elevación (0 a 8, lo que significa que las cámaras son 30, 35, 40, 45, 50, 55, 60, 65, 70 grados desde la horizontal respectivamente).
3. El acimut (0, 2, 4, ..., 34, multiplique por 10 para obtener el azimut en grados).
4. La condición de iluminación (0 a 5).

Para entrenamiento y pruebas regulares, los archivos "-dat" y "-cat" son suficientes. Los archivos "-info" se proporcionan en caso de que se necesiten otras formas de clasificación o preprocesamiento.

Los archivos se almacenan en el denominado formato de archivo "matriz binaria" o **binary matrix**, que es un formato simple para vectores y matrices multidimensionales de varios tipos de elementos. Los archivos de matriz binaria comienzan con un encabezado de archivo que describe el tipo y el tamaño de la matriz, y luego viene la imagen binaria de la matriz.

El número mágico codifica el tipo de elemento de la matriz:
- 0x1E3D4C51 for a single precision matrix
- 0x1E3D4C52 for a packed matrix
- 0x1E3D4C53 for a double precision matrix
- 0x1E3D4C54 for an integer matrix
- 0x1E3D4C55 for a byte matrix
- 0x1E3D4C56 for a short matrix

Los archivos "-dat" almacenan un tensor 4D de dimensiones 24300x2x96x96. Cada archivo tiene 24,300 pares de imágenes y cada imagen tiene 96x96 píxeles. Los archivos "-cat" almacenan un vector 2D de dimensión 24300x1, dado que cada par de imágenes comparten la misma categoría. Los archivos "-info" almacenan una matriz 2D de dimensiones 24300x4.

En la siguiente figura podemos ver diversas imágenes del conjunto de SmallNORB:

Figura 53. Muestras de SmallNORB.

A.3. Cifar10

El conjunto de datos CIFAR-10 consta de 60,000 imágenes en color de 32x32 en 10 clases, con 6,000 imágenes por clase. Hay 50,000 imágenes de entrenamiento y 10,000 imágenes de prueba.

El conjunto de datos se divide en cinco lotes de entrenamiento y un lote de prueba, cada uno con 10,000 imágenes. El lote de prueba contiene exactamente 1,000 imágenes seleccionadas al azar de cada clase. Los lotes de entrenamiento contienen las imágenes restantes en orden aleatorio, pero algunos lotes de entrenamiento pueden contener más imágenes de una clase que de otra. Entre ellos, los lotes de entrenamiento contienen exactamente 5,000 imágenes de cada clase.

Las imágenes de este *dataset* pueden pertenecer a 10 clases distintas: aviones, automóviles, pájaros, gatos, ciervos, perros, ranas, caballos, barcos y camiones.

El archivo descargable contiene cinco archivos data_batch (data_batch_1, data_batch_2, data_batch_3, data_batch_4 y data_batch_5) y un archivo test_batch. A su vez, cada uno de estos archivos por lotes contiene un diccionario con los siguientes elementos:

- data: una matriz numpy de 10000x3072 de uint8 s. Cada fila de la matriz almacena una imagen en color de 32x32. Las primeras 1024 entradas contienen los valores del canal rojo, las siguientes 1,024 el verde y las 1,024 finales el azul. La imagen se almacena en orden de fila principal, de modo que las primeras 32 entradas de la matriz son los valores del canal rojo de la primera fila de la imagen.
- labels: una lista de 10,000 números en el rango 0-9. El número en el índice i indica la etiqueta de la i-ésima imagen en los datos de la matriz.

El conjunto de datos contiene otro archivo, llamado batches.meta. También contiene un objeto de diccionario Python con la entrada label_names : una lista de 10 elementos que proporciona nombres significativos a las etiquetas numéricas en la matriz de etiquetas descrita anteriormente. Por ejemplo, label_names [0] == "avión", label_names [1] == "automóvil, etc.

En la siguiente figura podemos ver 10 ejemplos de cada una de las 10 clases que contiene el conjunto de cifar10:

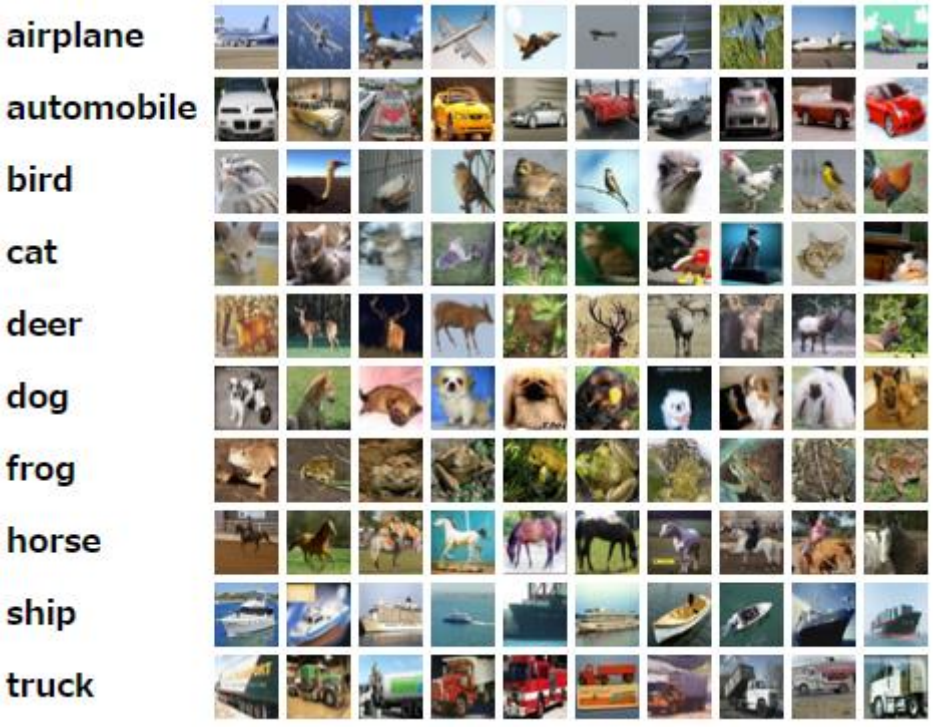

Figura 54. Muestras de Cifar10.

A.4. Retinografías

El conjunto total de datos consta de un conjunto de 484 retinografías. Este conjunto se puede dividir a su vez en 2 subconjuntos. Un primer subconjunto formado por 242 muestras tomadas a pacientes con la enfermedad ocular del glaucoma humano. Esta enfermedad se caracteriza por la pérdida de visión causada por un aumento de la presión ocular, dañando así el nervio óptico. Estas muestras han sido tomadas en el Hospital General Universitario Reina Sofía de Murcia.

El segundo subconjunto está formado por otras 242 muestras, esta vez realizadas a personas que no sufren dicha enfermedad. Este subgrupo se puede dividir a su vez en 2 grupos de imágenes. Un primer grupo formado por 132 muestras tomadas en el Hospital General Universitario Reina Sofía de Murcia. Un segundo grupo de 110 muestras que han sido tomadas por el Servicio de Oftalmología en el Hospital Miguel Servet, Zaragoza (España). Estas muestras han sido obtenidas de la base de datos de DRIONS. Empleamos este segundo grupo de muestras con el fin de balancear el número de imágenes no patológicas y el número de imágenes con la enfermedad de glaucoma.

Estos 2 grupos están separados en 2 carpetas diferentes. El nombre de cada imagen indica si se tiene o no dicha enfermedad. Cabe destacar que estas imágenes están en formato JPG y en color.

Las imágenes se adquirieron con una cámara de fondo de ojo analógica en color, aproximadamente centrada en el ONH y se almacenaron en formato de diapositiva. Para tener las imágenes en formato digital, se digitalizaron utilizando un escáner de alta resolución. A continuación, estas 124 muestras han sido recortadas quedándonos solo con la parte central de la retinografía, con un tamaño final de 50x50 pixeles.

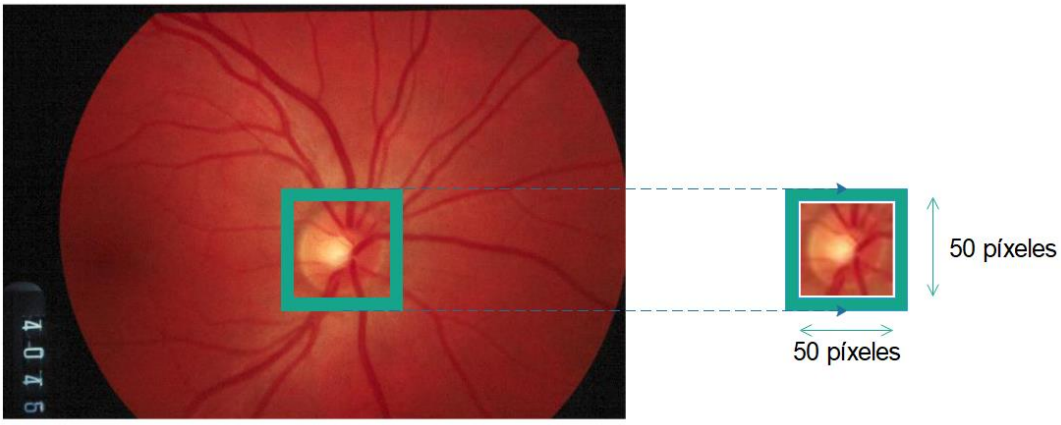

Figura 55. Preprocesado de retinografías.

B. Data augmentation

Esta es una práctica común a la hora de trabajar con redes neuronales. Se basa en el aumento de los datos presentes en el *dataset*. Suele ser bastante beneficioso emplear esta práctica cuando la proporción de clases no está balanceada y existen numerosas categorías con una única muestra. Para aumentar el número de datos disponibles para el entrenamiento se realizan una serie de procesados sobre una imagen original y así generar una serie de imágenes derivadas. Entre las posibles modificaciones existentes se encuentran la rotación, traslación, ruido, etc. Esto es efectivo dado que si modificamos una imagen, la red la percibe como una imagen totalmente nueva.

C. Tecnología empleada

C.1. Python

Usamos Python dado que es uno de los lenguajes más empleados y más estandarizados para trabajar en el campo de la inteligencia artificial. Esto es debido a que se han desarrollado multitud de librerías y *frameworks* especializadas en este ámbito. En nuestro caso usamos PyTorch la cual explicaremos a continuación.

C.2. PyTorch

Es un paquete de Python diseñado para realizar cálculos numéricos haciendo uso de la programación de tensores. Además permite su ejecución en GPU para acelerar los cálculos. Está desarrollado principalmente por el grupo de investigación de inteligencia artificial de Facebook.

PyTorch dispone una interfaz muy sencilla para la creación de redes neuronales pese a trabajar de forma directa con tensores sin la necesidad de una librería a un nivel superior como pueda ser Keras para Theano o Tensorflow. Aunque la interfaz de Python está más pulida y es el foco principal del desarrollo, PyTorch también tiene una interfaz C ++.

C.3. CUDA

Tras la aparición de las GPUs (*Graphics Processing Unit*) la inteligencia artificial se ha visto claramente beneficiada. Durante la ejecución de una red neuronal, especialmente en su etapa de entrenamiento, se realizan un gran número de cálculos y con una complejidad alta, lo cual hace necesario que empleemos un hardware de altas prestaciones.

Las GPUs son utilizadas en el entrenamiento de las redes neuronales dado que permiten reducir considerablemente los tiempos en comparación al uso exclusivo de CPUs. Esto es debido a su alto número de núcleos que permiten el procesamiento en paralelo de multitud de operaciones.

CUDA son las siglas de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo) y hacen referencia a una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA para la computación general en unidades de procesamiento gráfico (GPU). Esta herramienta es usada por Google Colab para realizar sus operaciones.

C.4. Google Colab

Es una herramienta de Google la cual nos permite ejecutar código de Python en la nube y crear modelos de *machine learning* empleando servicios de Google y con la posibilidad de usar GPU.

Esto nos proporciona dos ventajas muy significativas. La primera es que esta herramienta libera a nuestra máquina de tener que llevar a cabo un trabajo demasiado costoso tanto en tiempo como en potencia o incluso nos permite realizar ese trabajo si nuestra máquina no cuenta con recursos suficientemente potentes. Y todo de forma gratuita. La segunda ventaja es que facilita la colaboración dado que se pueden realizar tareas en la nube y posteriormente se pueden compartir los cuadernos (entornos) de trabajo.

C.5. Control de versiones GIT

Hemos empleado GIT como repositorio de código con el fin de poder guardar los distintos cambios realizados en el proyecto. Para ello, se han realizado 3 proyectos dentro de la plataforma GitHub, uno para MNIST y SmallNORB, dado que comparten la mayoría de hiperparámetros, otro para Cifar10 y otro para las retinografías. Esto facilitaba la incorporación de código en Google Colab dado que con una simple línea de código podemos clonar todo el proyecto en la plataforma. Las pruebas se han ido realizando directamente modificando archivos en Google Colab y los resultados finales han sido almacenados en *commits* o versiones del proyecto.

C.6. Edraw

Edraw es un software que nos ha sido útil para realizar los distintos diagramas y figuras realizados en este proyecto. En nuestro caso hemos usado la versión de pruebas 9.4. Esta es la última versión disponible de este software y, a pesar de tener menos funcionalidad que la versión de pago, nos ha dado un resultado esperado.

Referencias

Documentos

Sara Sabour, Nicholas Fross, and Geoffrey E Hinton (2017). *Dynamic routing between capsules. In Neural Information Processing Systems (NIPS).*

Sara Sabour, Nicholas Fross, and Geoffrey E Hinton (November 2017). *Matrix capsules with EM routing.*

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner (November 1998). *Gradient-based learning applied to document recognition.* Proceedings of the IEEE, 86(11):2278-2324.

Y. LeCun, F.J. Huang, L. Bottou, (2004) *Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting*. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR).

Dan C Cires¸an, Ueli Meier, Jonathan Masci, Luca M Gambardella, and Jurgen Schmidhuber. (2011) *High-¨performance neural networks for visual object classification*. arXiv preprint arXiv:1102.0183.

EJ Carmona, M. Rincón, J. García-Feijoo y JM Martínez-de-la-Casa (2008). *Identificación de la cabeza del nervio óptico con algoritmos genéticos. Inteligencia Artificial en Medicina, vol. 43 (3), págs. 243-259.*

Guo, Xifeng (December 2017), *CapsNet-Keras: A Keras implementation of CapsNet in NIPS2017 paper Dynamic Routing Between Capsules.*

Jonathan Hui. (December 2017), *Understanding Matrix capsules with EM Routing (Based on Hinton's Capsule Networks).*

Alex Krizhevsky (2009) *Learning Multiple Layers of Features from Tiny Images,*

Wei Zhao , Jianbo Ye , Min Yang , Zeyang Lei , Soufei Zhang , Zhou Zhao (Sptember 2018) *Investigating Capsule Networks with Dynamic Routing for Text Classification.*

Jorge Larrey, (2018) *INTRODUCCION AL MACHINE LEARNING.*

B. Mandal, S. Dubey, S. Ghosh, R. Sarkhel and N. Das, (2018) Handwritten Indic Character Recognition using Capsule Networks, IEEE Applied Signal Processing Conference (ASPCON), Kolkata, India, 2018, pp. 304-308.

Atefeh Shahroudnejad, Arash Mohammadi, Konstantinos N. Plataniotis (February 2018) *Improved Explainability of Capsule Networks: Relevance Path by Agreement*.

Rodney LaLonde, Ulas Bagci (April 2018) *Capsules for Object Segmentation.*

Simon Haykin (2008) *Neural Networks and Learning Machines*, Third Edition

Enlaces

https://pytorch.org/docs/stable/index.html

https://es.stackoverflow.com/

https://github.com/yl-1993

https://github.com/ahmedshoaib

https://colab.research.google.com

https://towardsdatascience.com

https://medium.com

https://openreview.net

https://www.researchgate.net