Reconocimiento de caracteres utilizando Capsule Nets

Ancor López Hernández

Resumen– El objetivo de este proyecto es, mediante la utilización del nuevo concepto de red neuronal Capsule Nets, conseguir reconocer caracteres sobre diferentes datasets, para ello primero se ha realizado un experimento sobre MNIST, ya que es sobre el que se trabaja en el paper original y se conoce el estado del arte actual, seguidamente se ha probado la red sobre otro dataset de letras y finalmente se han realizado dos experimentos sobre datasets propios creados de bigramas y trigramas. Los resultados obtenidos durante este proyecto demuestran que Capsule Nets es capaz de llegar al estado del arte, además de poder reconocer varios caracteres en una misma imagen.

Palabras clave– Reconocimiento de caracteres, deep learning, capsule nets, routing by agreement, pytorch

Abstract– The aim of this project is, through the use of the new Capsule Nets neural network concept, to recognize characters on different datasets, in order to do this, first of all an experiment on MNIST has been carried out, since it is the one referenced in the original paper and on which the state of current art is known, then the network has been tested on another dataset of letters and finally two experiments have been carried out on own datasets created on bigrams and trigrams. The results obtained during this project show that Capsule Nets is able to reach the state of the art, as well as being able to recognize different characters in the same image.

Keywords– Character recognition, deep learning, capsule nets, routing by agreement, pytorch

1 Introducción

Hoy en día hay muchas maneras de hacer Visión por Computador, la cual trata de reproducir el funcionamiento de los ojos y cerebros humanos para los ordenadores puedan captar y comprender una imagen y actuar según convenga. Uno de los modelos computacionales que se utilizan en este campo son las redes neuronales cuya idea principal es inspirarse en la arquitectura biológica, cómo funciona el cerebro y cómo se comunican las neuronas para intentar reproducir toda esa arquitectura y comportamiento de manera digital. En este proyecto se pretende utilizar la red neuronal Capsule Nets [1] recientemente publicada. Se ha demostrado que esta red es capaz llegar al estado del arte conseguido por otras redes neuronales sobre MNIST [2]. Otro aspecto interesante es que Capsule Nets tiene una arquitectura diferente a las demás CNNs, siendo más parecida a como deberían estructurarse y funcionar nuestras neuronas. Como se verá en la sección 2, se va a probar el código que ya está implementado en la web [3].

Para seguir aprendiendo y explorando las posibilidades de Capsule Nets, se entrenará sobre datasets de N-gramas, ya que una hipótesis es que, si es capaz de obtener buenos resultados reconociendo diferentes caracteres en la misma imagen, se podría utilizar Capsule Nets para hacer Word Spotting: la imagen de la palabra se codifica en alguna manera que se pueda comparar con un query dado normalmente en forma textual. Se verá más en detalle en la sección 3.3.

Todo esto con el objetivo principal de ver cuales son las ventajas y limitaciones que Capsule Nets nos ofrece respecto a las CNNs convencionales.

2 Objetivos

Con el fin de ver todas las ventajas/desventajas de Capsule Nets, se han marcado una serie de objetivos divididos en dos tipos: obligatorios y optativos.

Objetivos obligatorios: Se han decidido que fuesen obli-

• E-mail de contacto: ancorlh@gmail.com
• Mención realizada: Computación
• Trabajo tutorizado por: Dimosthenis Karatzas (CVC)
• Curso 2017/18

gatorios, los objetivos que se ha considerado que son la base necesaria para empezar con Capsule Nets. De todos estos objetivos se tiene código ya implementado y datasets creados.

Objetivos optativos: Estos objetivos son de carácter exploratorio. Para poder desarrollarlos, primero se necesita haber realizado todos los obligatorios. Por lo tanto, son más difíciles de planificar desde el principio del proyecto y conllevan un riesgo mayor que los obligatorios.

1. [OBLIGATORIO] Aprender Pytorch, ya que es el framework que se utilizará para implementar esta red neuronal.
2. [OBLIGATORIO] Entender, ejecutar el código y obtener resultados de una de las implementaciones de la red neuronal CapsuleNets [3] sobre el dataset MNIST.
3. [OBLIGATORIO] Utilizar otro dataset adaptando el código ya existente, en nuestro caso utilizaremos EMNIST-letters, para ver cómo se comporta nuestra red.
4. [OPTATIVO] Desarrollar software necesario para poder crear un dataset de bigramas y adaptar la arquitectura de Capsule Nets para ver si es capaz de reconocerlos.
5. [OPTATIVO] Generar un último dataset de Trigramas, adaptar la arquitectura para que pueda reconocer dos bigramas dentro de cada trigrama.

3 Estado del Arte

La clasificación de imágenes es un problema central en Machine Learning, es por esto por lo que surgen las Redes Neuronales Convolucionales (CNN). La idea básica de estas redes es entrenarse con imágenes etiquetadas para que el modelo pueda ir abstrayendo las características (features) que le permitan clasificar correctamente la imagen. Una vez entrenada se podrá utilizar ese modelo para clasificar nuevas imágenes sin etiquetar. A lo largo de los años, estas redes han llegado al estado del arte por introducir nuevos conceptos como veremos más adelante.

MNIST ha sido durante muchos años un dataset de referencia en el campo, y aunque saturado, sigue siendo un benchmarking aceptado por la comunidad internacional. Por ello, las principales CNNs lo utilizan como referencia. El mejor resultado obtenido sobre MNIST lo ha conseguido la red de Wan [4], con un 0.21 % de test error y pre procesando las imágenes. Capsule Nets ha conseguido un 0.25 % de test error sin ningún tipo de pre procesamiento, solamente conseguido antes por redes neuronales mucho más profundas.

A modo de pequeño repaso y puesta en contexto se muestran algunas de las redes neuronales más importantes:

3.1. Breve historia de las redes neuronales

3.1.1. LeNeT-5

Fue una CNN de 7 capas pionera introducida por LeCun en 1988 [5] y fue utilizada por la gran mayoría de bancos para clasificar los dígitos escritos a mano en los cheques. Una de sus claves para el éxito fue utilizar convoluciones para extraer características espaciales y fue la base sobre las que se desarrollaron las siguientes arquitecturas.

3.1.2. AlexNet

En 2012, Alex Krizhevsky [6] presentó esta red que introducía nuevas mejoras como el uso de ReLU que ayudó a prevenir *The vanishing gradient problem* [3] e introdujo el concepto de *dropout* que consiste en activar y desactivar aleatoriamente las neuronas de cada capa para prevenir el overfitting. También introdujo el concepto de *data augmentation* que consiste en entrenar a la red neuronal con imágenes con diferente rotación y ángulo.

3.1.3. VGGNet

Fue presentada en 2014 por Karen Simonyan and Andrew Zisserman [7], su principal mejora fue añadir más capas a la red neuronal.

3.1.4. GoogleNet

Esta red [8] creada en 2015 permite que en cada capa se puedan realizar operaciones convoluciones y pooling a la vez concatenado sus salidas y propagándolas a la siguiente capa lo que se traduce en un mejor aprendizaje de las features en cada capa.

3.1.5. Microsoft ResNet

Creada en 2015 [9], la idea detrás de esta red neuronal es que, en cierto punto, añadir más capas no mejora el rendimiento de la red, de hecho, ocurre todo lo contrario debido al gradiente. Para evitar esto cada dos capas un *identity mapping addition* lo cual ayuda a propagar el error.

Con todos estos avances, las CNN han conseguido obtener excelentes resultados en materia de reconocimiento de objetos en imágenes. Pero para ello necesitan una gran cantidad de muestras para realizar el entrenamiento. Esto es debido a que las CNNs no codifican de manera explícita la posición concreta y relaciones espaciales entre características relevantes para la tarea de clasificación, sino detectan su presencia en la imagen. Es aquí donde entra Capsule Nets para tratar de solventar estos problemas.

3.2. MNIST

A lo largo de los años el dataset MNIST [2] se ha convertido en un punto de referencia para probar diferentes métodos de clasificación como CNN o Máquinas de Vectores de soporte (SVM). El principal motivo del uso de MNIST es que contiene una gran cantidad de muestras, 60000 para training y 10000 para testing (70000 en total), de dígitos escritos a mano, los cuales se han normalizado y centrado en la imagen.

Para poder comparar el rendimiento de Capsule Nets en MNIST, se utilizará el Top 7 de CNNs sin hacer preprocesamiento de las imágenes, siendo un 0.35 % el mejor re-

sultado. La tabla completa con todas las CNNs se puede encontrar en la web oficial de MNIST [10].

3.3. Word Spotting

Cuando no se puede hacer reconocimiento, se hace word spotting, introducido previamente en la sección 1.

Esta técnica se utilizó primeramente sobre documentos manuscritos. En los últimos 5 años, se utiliza para detectar en texto en imágenes de escenas.

Muchos de los métodos usados para representar texto (palabras) parten de un diccionario predefinido. Por ejemplo, Word2Vec [11] define un espacio donde palabras sistemáticamente similares están más cerca. Otros métodos, como por ejemplo PHOC, codifican las palabras en base a su distribución de caracteres. Para conseguir esto, PHOC utiliza histogramas. Se pueden entrenar redes neuronales para que produzcan una representación de PHOC dada una imagen de la palabra, como por ejemplo PHOCnet [12]. Capsule Nets podrían ofrecer maneras alternativas para detectar la existencia y relaciones de caracteres, y n-gramas en imágenes de texto.

4 CAPSULE NETS

Esta red neuronal [1] se divide en 2 partes: Encoder y Decoder. Las primeras 3 capas forman parte del Encoder y las 3 siguientes forman el Decoder.

4.1. Encoder

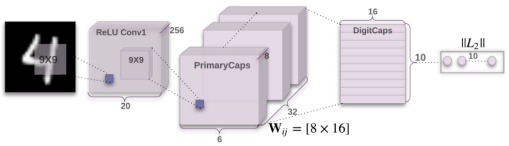

Fig. 1: Arquitectura Encoder de Capsule Nets [1]

4.1.1. Capa 1 - Convolutional Layer

Detecta las basic features de nuestras imágenes de entrada. Consta de 256 feature maps con kernels de 9x9x1 y stride 1 con función de activación ReLU.

4.1.2. Capa 2 - Primary Caps

Cada Primary Caps está formada por una matriz de 6x6 capsulas de vectores de 8 Dimensiones, las cuales tienen 32 maps o Primary Caps, en total tendremos 1152 capsulas (6x6x32) con kernels de convolución de 9x9x256.

4.1.3. Capa 3 - DigiCaps

Está formada por 10 DigitCapsules, una por cada número que queremos reconocer, con un vector de output de 16 Dimensiones. Como reciben el vector de 8Dimensiones de las PrimaryCaps, cada uno de estos vectores tiene su matriz de pesos de 8x16 para transformar los vectores 8D en 16D.

Para calcular el *Loss* de la red se utilizará la siguiente función: Esto quiere decir que, si un objeto de la clase k

$$L_k = T_k \ \max(0, m^+ - ||\mathbf{v}_k||)^2 + \lambda \ (1 - T_k) \ \max(0, ||\mathbf{v}_k|| - m^-)^2$$

Fig. 2: Función Loss [1]

está presente en la imagen, la capsula correspondiente de nivel más alto tendrá un output vector cuya norma será como mínimo de 0.9, de lo contrario el output vector tendrá una Norma inferior a 0.1.

4.2. Decoder

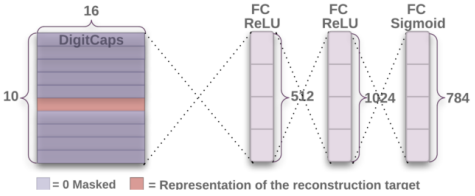

Fig. 3: Arquitectura Decoder de Capsule Nets [1]

Su función es coger el vector de 16D que tiene como output el Encoder, aplicarle una máscara para dejar solo el vector con el output más grande y aprender a decodificarlo como la imagen de un número y recrear una imagen de 28x28 como la que teníamos en la entrada de Encoder. Como se puede observar en la figura 3, consta de tres capas fully connected.

4.3. Routing by Agreement

Una de las novedades introducidas Capsule Nets, en vez de utilizar métodos de pooling como la gran mayoría de redes neuronales, es este algoritmo el cual utilizan las capsulas para modificar sus pesos y determinar hacia que capsula del nivel superior envían sus outputs.

Procedure 1 Routing algorithm.

1: **procedure** ROUTING($\hat{\mathbf{u}}_{j|i}, r, l$)
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 $\mathbf{v}_j$

Fig. 4: Algoritmo de Routing by Agreement [1]

Por contrapartida, el tener el algoritmo de Routing en vez de hacer pooling, hará que nuestro proceso de aprendizaje sea bastante más lento, ya que como se puede ver en la figura 4, se tienen que computar más operaciones y además hay que iterar sobre cada cápsula.

4.4. ¿Cómo funcionan las capsulas?

Capsule vs. Traditional Neuron			
Input from low-level capsule/neuron		vector($\mathbf{u}_i$)	scalar(x_i)
Operation	Affine Transform	$\widehat{\mathbf{u}}_{j\|i} = \mathbf{W}_{ij}\mathbf{u}_i$	–
	Weighting	$\mathbf{s}_j = \sum_i c_{ij}\widehat{\mathbf{u}}_{j\|i}$	$a_j = \sum_i w_i x_i + b$
	Sum		
	Nonlinear Activation	$\mathbf{v}_j = \frac{\|\mathbf{s}_j\|^2}{1+\|\mathbf{s}_j\|^2}\frac{\mathbf{s}_j}{\|\mathbf{s}_j\|}$	$h_j = f(a_j)$
Output		vector($\mathbf{v}_j$)	scalar(h_j)

Fig. 5: Diferencias entre capsulas y neuronas Fuente:[13]

En la figura 5 se puede ver como las capsulas funcionan con vectores en vez de escalares, esto les permite tener más capacidad representativa. También se puede observar como introduce la multiplicación matricial de pesos *W*, que codifica información importante de la relación espacial entre características de bajo y alto nivel, por ejemplo, podría codificar la relación entre un ojo (bajo nivel) y una cara (alto nivel). Otro aspecto importante es que desaparece el bias a la hora de hacer el sumatorio de las capsulas y la utilización de una función de activación no linear que aplica un squash a los vectores para que su longitud no sea mayor de 1, preservando la dirección de éstos.

4.5. Capsule Nets con EM Routing

En [14], se introduce una nueva versión de cápsula en la cual está formada por una unidad logística para representar la presencia de una entidad y una matriz 4x4 para representar la relación entre la entidad y el observador (pose). También se introduce el algoritmo de Expectation-Maximization al proceso de routing para que el output de cada cápsula sea direccionado a la cápsula de nivel superior cuyo cluster tenga votos similares.

5 Metodología

Primeramente, se ha aprendido a utilizar PyTorch mediante los tutoriales proporcionados en su página web. Una vez hecho esto, se ha ejecutado el código de [3] donde está implementada la red Capsule Nets. Se han obtenido los primeros resultados y se ha analizado y comparado con el Top 7 el accuracy/error que se consigue sobre el dataset MNIST. También se ha analizado la reconstrucción del Decoder y modificado las dimensiones del output para observar los cambios en los dígitos.

Para poder ejecutar el código se han necesitado las siguientes tecnologías/librerías:

- Python 3 ya que la red neuronal Capsule Nets está implementada en este lenguaje.
- Pytorch, comentado anteriormente en 2.
- TorchVision para poder descargar y acceder a los datasets sobre los cuales entrenaremos y evaluaremos nuestra red neuronal además de permitir la opción de aplicar transformaciones a las imágenes como el pasarlas a gris o rotarlas.
- Numpy para la gestión de matrices en CPU.
- Matplotlib para poder visualizar por pantalla los resultados obtenidos de nuestra red neuronal y poder compararlos con los del dataset original.

Uno de los problemas surgidos a la hora de ejecutar el código es que las GPUs de las que dispongo no son compatibles con la tecnología CUDA, por lo tanto, se ha utilizado Google Colaboratory.

Para el siguiente experimento, se ha ejecutado el código y analizado los resultados sobre otro dataset, en este caso EMNIST-letters, para ver cómo se comporta la red con más clases y sobre caracteres en vez de dígitos.

A continuación, como siguiente experimento, se ha generado un dataset de bigramas de letras en inglés propio. Se han hecho las modificaciones en la arquitectura de la red neuronal pertinentes para adaptarla y se han analizado sus resultados.

Para finalizar, se ha generado un último dataset de trigramas. Se ha vuelto a modificar la red la red a éste dataset y se han analizado sus resultados.

Para la planificación de este proyecto se ha utilizado un Diagrama de Gantt, dónde se ha puesto las tareas a realizar con el tiempo que llevará cada una de ellas, se puede ver en A.1.

6 Reconocimiento de dígitos sobre sobre MNIST

Para la realización del primer experimento de ejecutar el código de Capsule Nets sobre el dataset de MNIST [2]. Se ha descargado el código de Capsule Nets e instalado todas las librerías y frameworks necesarios para su correcto funcionamiento. Como estamos utilizando una versión de Capsule Nets implementada en PyTorch, se ha utilizado la documentación oficial de su página web [15] para aprender y entender su funcionamiento.

Una vez entendido todo, se ha procedido a ejecutar el código, ver qué resultados daba y si estos eran los esperados o no. Como se ha comentado en 5, se ha usado Google Colaboratory para realizar el entrenamiento de Capsule Nets. Colaboratory permite ejecutar una Jupyter Notebook de Python en una máquina virtual que Google te asigna y así poder disponer de una GPU para acelerar la ejecución de nuestro código. Se han ejecutado 20 epochs ya que el tiempo del cual dispones de la máquina virtual es limitado. Los demás parámetros de la red neuronal se han dejado como el paper indica y el batch utilizado es de 100.

6.1. Resultados

TABLA 1: RESULTADOS TRAINING Y TEST

	Accuracy %	Error %	Loss
Train	99.95	0.048	0.0045
Test	99.41	0.59	0.019

TABLA 2: TOP 7 CNN VS CAPSULE NETS EN MNIST

Nombre de la CNN	**Error %**
Capsule Nets (paper) [1]	0.25
large/deep conv. net, [elastic distortions]	0.35
large/deep conv. net, unsup. pretraining [elastic distortions] [16]	0.39
Convolutional net, cross-entropy [elastic distortions]	0.4
large conv. net, unsup pretraining [no distortions] [17]	0.53
Trainable feature extractor + SVMs [affine distortions] [18]	0.54
Trainable feature extractor + SVMs [elastic distortions] [18]	0.56
unsupervised sparse features + SVM, [no distortions] [19]	0.59
Capsule Nets (implementaicón usada)	0.59

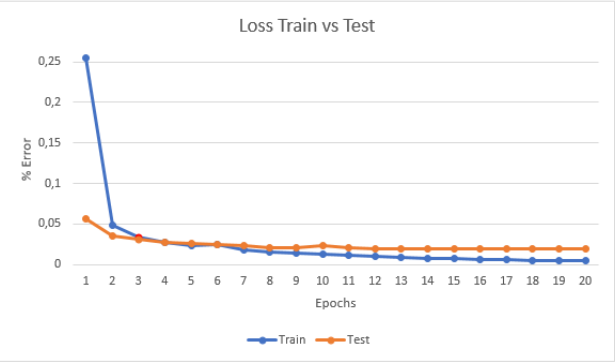

Fig. 6: MNIST Test vs Training Loss

Como se puede observar en la tabla 1, tanto en el set de training como el de accuracy se consigue llegar a un accuracy prácticamente perfecto (99.95 % en training y 99.41 % en test). Además de obtener un error de un 0.59 % en el set de Test en las 20 epochs ejecutadas. El estado del arte sobre MNIST es del 0.21 % comentado previamente en 3. Teniendo en cuenta que en [1] se consigue un error del 0.25 % al final de toda la ejecución y observando la tendencia del Total Loss en la figura 6, se puede decir que nuestro 0.59 % de error hubiera llegado a acercarse a ese 0.25 % si se hubieran ejecutado más epochs.

En la tabla 2 se puede observar que la implementación utilizada está dentro del Top 7 sobre MNIST. Este resultado no es malo aunque se esperaba conseguir un error más bajo y poder estar en el Top 3.

6.1.1. Comparación input - output

Fig. 7: Input Original

Fig. 8: Reconstrucción output

Como se puede observar en la figura 8, la reconstrucción de los dígitos es correcta e incluso más legible. También se puede observar que los dígitos son más gruesos en la reconstrucción que en el input original. Esto es debido a que se ha modificado una de las 16 dimensiones de los vectores output de DigiCaps. Para demostrar cómo influye cada dimensión en aspectos de la reconstrucción, se ha cogido un dígito y se ha modificado cada una de las dimensiones en un rango de -0.25 a 0.25 con un intervalo de 0.05.

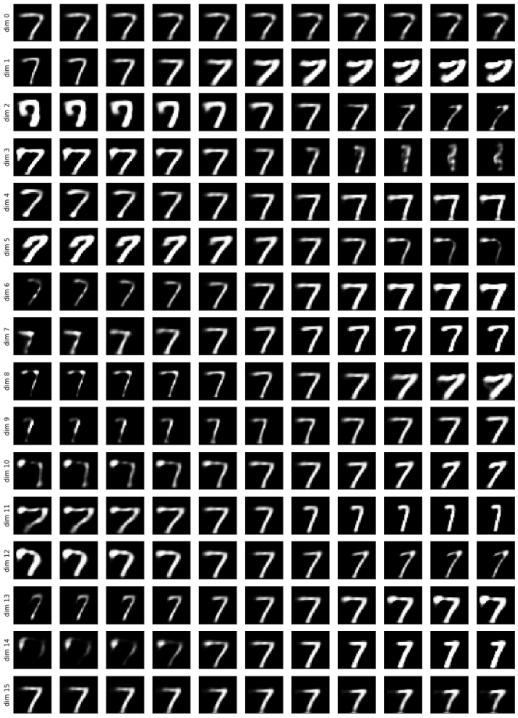

Fig. 9: Variación dígito según dimensión

Por lo tanto, se puede comprobar en la figura 9 como cada una de estas dimensiones controlan características de los dígitos como el grosor, la altura o la curvatura por ejemplo.

7 RECONOCIMIENTO LETRAS SOBRE EMNIST-LETTERS

Para este experimento, se ha escogido el dataset de EMNIST que contiene letras para ver cómo de bien clasifica ca-

racteres Capsule Nets. Este dataset contiene 26 clases, una por cada letra del alfabeto inglés. Se ha procedido a modificar la arquitectura de Capsule Nets haciendo que la capa de DigiCaps contenga 26 DigiCapsules para poder almacenar todas las letras. También se ha modificado la entrada del Decoder para poder recibir estos 26 vectores para su posterior reconstrucción. Además, se han añadido nuevas clases para poder leer el nuevo dataset y hacer las particiones de train/test.

En una primera ejecución, se observó que las imágenes de las letras que se pasaban como input al Encoder parecían estar rotadas y tener alguna modificación geométrica más. Por este motivo, se decidió aplicar un flip horizontal y una rotación en ángulo de 90ª a cada imagen. Como se verá en 7.0.2, ahora sí que se muestran correctamente las letras.

Al constar de 88.800 muestras para training y 14.800 para el test (103.600 en total), este dataset es mayor que MNSIT (70.000 en total) y al casi triplicar el número de clases, 26 contra 10, se espera que el entrenamiento lleve aún más tiempo que para el experimento anterior. Por lo tanto, tan solo se han ejecutado 5 epochs, ya que Google Colaboratory asigna las GPUs por un tiempo limitado. El resto de parámetros se han dejado igual que en el experimento anterior.

7.0.1. Resultados

TABLA 3: RESULTADOS TRAINING Y TEST

	Accuracy %	Error %	Loss
Train	94.32	5.67	0.054
Test	93.57	6.42	0.059

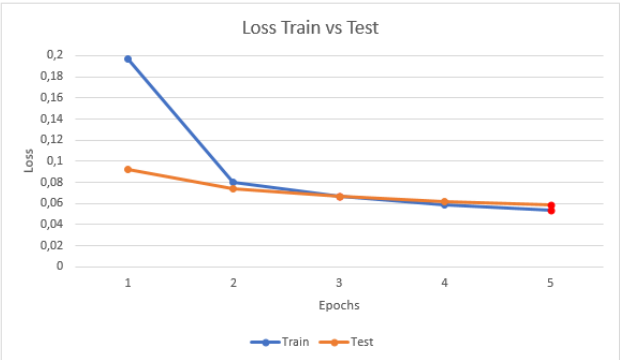

Fig. 10: Test vs Train Loss en EMNIST-letters

A falta de poder ejecutar más epochs por falta de tiempo, se puede observar en la tabla 3 que el accuracy máximo en el set de Test es de 93,57 % y 94,32 % en Training. Como en 6, si observamos la tendencia del Loss en la figura 10, aumentando el número de epochs, se hubiesen conseguido mejores resultados sobre EMNIST-letters.

7.0.2. Comparación letras input - output

Fig. 11: Input letra original

Fig. 12: Reconstrucción output

Al igual que en 6.1.1 se puede observar como en la reconstrucción las letras, éstas tienen un grosor superior a las de entrada. También se puede ver que se han realizado las transformaciones geométricas adecuadas para la correcta visualización de nuestras imágenes.

8 RECONOCIMIENTO DE N-GRAMAS

8.1. Reconocimiento de Bigramas

Para la realización de este experimento se ha creado un script en Python que genera imágenes de los 20 bigramas más utilizados en inglés. Se han utilizado todas las fuentes que tiene el sistema Linux, distribución Ubuntu. También se ha cambiado el tamaño de 28 x 28 pixeles a 48 x 48 ya que ahora nuestra imagen contiene dos letras.

Además, se ha puesto el bigrama en una posición aleatoria, tanto en altura como en anchura, dentro de cada imagen. Con todo esto se consigue un dataset de 88.593 imágenes, con 70.976 de training y 17.617 para el test. Se han modificado los siguientes aspectos en la arquitectura de nuestra red neuronal:

1. El número de clases pasa a ser 20, una por cada bigrama.

2. Se ha reducido el tamaño del batch a 50 ya que al ser imágenes con mayor número de pixeles no había suficiente memoria para poder ejecutar nuestro código.

3. Se ha modificado el input de DigiCaps de acuerdo con el nuevo output del punto 3, [6x34x32].

4. Por último, se ha modificado las 3 capas Fully Connected del Decoder para generar la imagen reconstruida de 28x84.

Al aumentar todos estos parámetros, hace que nuestro algoritmo vaya más lento tan solo hemos podido ejecutar un total de 3 epochs.

8.1.1. Resultados

TABLA 4: RESULTADOS TRAINING Y TEST

	Accuracy %	Error %	Loss
Train	94.53	5.46	0.12
Test	89.26	10.74	0.195

Como se muestra en la tabla 4 de resultados del test, se consigue llegar a un accuracy del 89,26 %. Se puede decir que es aceptable y que probablemente, con la evolución descendente en la figura 13 del loss, tanto de train como de test, aumentando el número de epochs se habría llegado a unos resultados mejores.

Fig. 13: Test y Training Loss

8.1.2. Comparación bigramas input - output

Fig. 14: Input bigrama original

Fig. 15: Reconstrucción output

Se puede observar como en la reconstrucción del bigrama hay un difuminado, esto también se observa en [20] y se puede atribuir probablemente a la falta de información para la correcta reconstrucción de la imagen. Como se menciona en [1], al ejecutar Capsule Nets sobre el dataset de Cifar10, han modificado la arquitectura para que en vez de 32 capsulas, hayan 64 y así poder recoger mejor toda la información de las imágenes. Lo siguiente que se observa en la reconstrucción, es que el bigrama siempre aparece en el centro de la imagen, cuando debería aparecer en la misma posición donde se encuentra el original. No se ha conseguido deducir una razón clara por la que esto ocurra.

8.2. Reconocimiento de Trigramas

Para la realización de este último experimento, se ha generado un dataset con un total de 16 trigramas diferentes. Cada trigrama está formado por dos bigramas ya que la finalidad del experimento es reconocer estos bigramas. Se tendrá un total de 25 clases, con lo cual el conjunto de train contará con 9,072 imágenes y el de test con 2,112. Como solo se han cogido 16 trigramas, el número de bigramas está restringido al no poder utilizarse todas las combinaciones posibles. Además, como cada trigrama pertenece a 2 clases, se ha generado un archivo .csv con el nombre de la imagen y las clases a las que pertenece.

Al tener un dataset diferente a los vistos anteriormente, se ha tenido que modificar la clase base que los cargaba en nuestra red para que, a cada muestra de los sets, le asignase las dos etiquetas de sus clases correspondientes. También se ha tenido que modificar la arquitectura para adaptarla a las 25 clases y detectar las dos cápsulas con vectores de mayor valor en el output que nos da la capa de DigiCaps. Al hacer esto, nos surge la limitación de que, si hubiese un trigrama formado por *aaa*, suponiendo que el bigrama *aa* pertenece a la clase 1, el output correcto que debería tener es (02000000000000000000000), pero como sacamos los dos vectores de mayor valor, solo nos daría la primera clase a la que pertenece en vez de decir que hay dos bigramas de la misma clase.

Como estas nuevas imágenes tienen más información en ellas, se ha tenido que reducir el batch a 25 porque el sistema se quedaba sin memoria realizando el entrenamiento. Por este motivo, la velocidad de ejecución de cada epoch ha descendido y solo se ha ejecutado un total de 5. Al ver la mala reconstrucción realizada en la figura15, se ha optado por no realizar reconstrucción en este experimento, ya que se ha creído que lo importante era ver si Capsule Nets era capaz de clasificar correctamente estos trigramas.

A continuación, se muestra una imagen de ejemplo para tener una idea del aspecto que tiene nuestro input:

Fig. 16: Input trigrama original

8.2.1. Resultados

TABLA 5: RESULTADOS TRAINING Y TEST

	Accuracy %	Error %	Loss
Train	99.08	0.82	0.052
Test	98.00	2.00	0.096

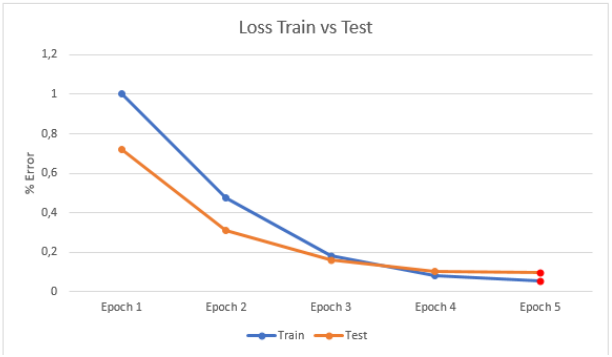

Fig. 17: Test y Train Loss sobre Trigramas

De la tabla 5, aun siendo sobre un dataset con pocas muestras, se puede decir que Capsule Nets es capaz de reconocer y clasificar correctamente más de un objeto en una misma imagen. El mejor accuracy conseguido al finalizar la ejecución es de un 98.00 % en el dataset de test. Además, si se observa como desciende el loss en la figura17, esta arquitectura podría llegar a un accuracy superior.

A modo de ilustración, en la figura 18 , se puede observar como para el trigrama *for*, el output del Encoder con las 25 clases, nos enmascara correctamente las posiciones de las clases a las cuales pertenece este trigrama. Por lo tanto, se deduce que es capaz de diferenciar el bigrama *fo* y *or*.

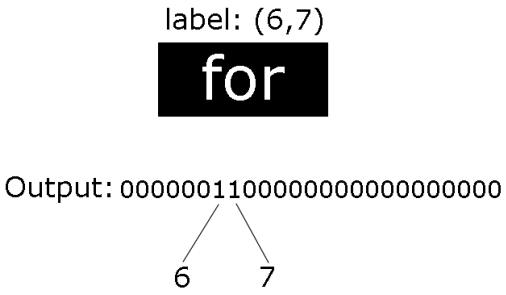

Fig. 18: Output Encoder Trigramas

9 Conclusiones

Como se ha podido comprobar con el experimento sobre MNIST 6.1 y EMNIST-letters 7.0.1, Capsule Nets consigue excelentes resultados en cuanto al reconocimiento y reconstrucción de caracteres. Consigue llegar a un 0,59 % de error y seguramente con capacidad de mejora si se hubiese entrenado por más tiempo. Gracias a la obtención de estos resultados puedo decir que se han realizado satisfactoriamente los objetivos obligatorios.

Además, de los resultados obtenidos sobre nuestro dataset de bigramas 8.1.1, se puede concluir que nuestra red es capaz de detectar conjuntos de caracteres agrupados. También se puede deducir de la reconstrucción que nos ha hecho el Decoder en estos bigramas, que cuánta más información contenga la imagen sobre la que estamos trabajando, necesitaremos tener más capas de cápsulas o más cápsulas en cada capa para conseguir almacenar toda la información necesaria. Con esto, se conseguirá una correcta reconstrucción y que desaparezca ese efecto de difuminado. Sobre el problema de que cuando reconstruimos el bigrama nos salga siempre centrado, después de investigar, no se ha podido dar una explicación/solución.

Respecto al experimento sobre trigramas, aunque ejecutado con un dataset pequeño, se puede afirmar que Capsule Nets es capaz de detectar más de una clase en una misma imagen. Con esto se confirma la hipótesis que se planteaba en 1, se podría llegar utilizar esta red neuronal en el campo de Word Spotting para la detección de palabras.

Una conclusión muy interesante es que como cada cápsula guarda relación espacial entre las diferentes features, Capsule Nets necesita una fracción de muestras para entrenarse comparada con las CNNs clásicas. Así se evita entrenar a Capsule Nets para aprender todas las variaciones que pueden tener los objetos que se quieren reconocer y clasificar.

Para acabar, de todos los experimentos realizados he concluido que ejecutar esta red en CPU es prácticamente inviable ya que la complejidad computacional que añade el algoritmo de routing es elevada. Esto hace que se tarde unas 3 horas en realizar una epoch sobre MNIST en CPU. Utilizando una GPU pasa a tarde 30 minutos en realizar cada epoch. De aquí que el autor de Capsule Nets, Geoffrey Hinton, propusiese éste modelo hace tiempo y por falta de potencia de cómputo en las GPUs se ha tenido que esperar hasta ahora para poder implementarla.

10 Agradecimientos

Quisiera agradecer a mi tutor Dimosthenis Karatzas, ya que sin su ayuda y consejos este proyecto no se habría podido realizar.

Referencias

[1] N. Frosst S. Sabour and G. Hinton. *Dynamic Routing Between Capsules*, 2017.

[2] Y. Bengio Y. LeCun, L. Bottou and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, vol. 86,:pp. 2278–2323, 1998.

[3] Dulat Yerzat. *Capsule-Network-Tutorial*, 2017.

[4] S. Zhang Y. LeCun L. Wan, M. Zeiler and R. Fergus. *Regularization of Neural Networks using DropConnect*, 2013.

[5] Y. Bengio Y. LeCun, L. Bottou and P. Haffner. *Gradient-Based Learning Applied to Document Recognition*, 1988.

[6] G. Hinton A. Krizhevsky and I. Sutskever. *ImageNet Classification with Deep Convolutional*, 2012.

[7] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.

[8] Y. Jia P. Sermanet S. Reed D. Anguelov D. Erhan V. Vanhoucke A. Rabinovich Google Inc University of North Carolina C. University of Michigan C. Szegedy,

W. Liu and Ann Arbor. *Going Deeper with Convolutions*, 2015.

[9] S. Ren K. He, X. Zhang and J. Sun. *Deep Residual Learning for Image Recognition*, 2015.

[10] C. Cortes Y. LeCun and C. Burges.

[11] K. Chen T. Mikolov, G. Corrado and J. Dean. *Efficient Estimation of Word Representations in Vector Space*, 2013.

[12] S. Sudholt and G.A. Fink. *PHOCNet: A deep convolutional neural network for word spotting in handwritten documents*, 2016.

[13] Max Pechyonkin. *Understanding Hinton's Capsule Networks. Part II: How Capsules Work*, 2017.

[14] S. Sabour G. Hinton and N. Frosst. *Matrix Capsules with EM Routing*, 2018.

[15] PyTorch. *Sitio web oficial de PyTorch*, 2017.

[16] S. Chopra M. Ranzato, C. Poultney and Y. LeCun. *Efficient Learning of Sparse Representations with an Energy-Based Model*, 2006.

[17] M. Ranzato K. Jarrett, K. Kavukcuoglu and Y.LeCun. *What is the Best Multi-Stage Architecture for Object Recognition?*, 2003.

[18] C. Suen F. Lauer and G. Bloch. *A trainable feature extractor for handwritten digit recognition*, 2007.

[19] E. Barth K. Labusch and T. Martinetz. *Simple Method for High-Performance Digit Recognition Based on Sparse Coding*, 2008.

[20] theblackcat102. *dynamic-routing-capsule-cifar*, 2017.

APÉNDICE

A.1. Sección del Apéndice

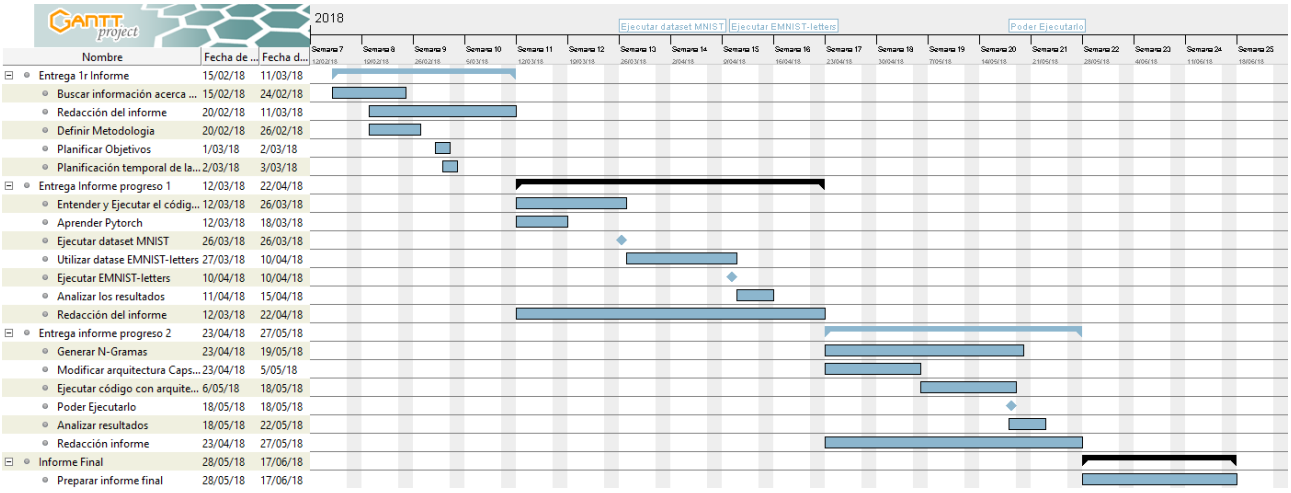

Fig. 19: Diagrama Gantt de la planificación del proyecto