Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Cálculo de la tensión de Von Misses con redes neuronales convolucionales

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Martín Barroso Ordóñez

Co-tutor: Carlos Monserrat Aranda

Co-tutor: María José Rupérez Moreno

2020-2021

Resumen

Las redes neuronales se emplean ampliamente para la resolución de problemas complejos a velocidades nunca vistas. Hemos extraído muestras de entrenamiento para el cálculo de tensiones de Von Misses para estructuras en voladizo, buscado los mejores hiperparámetros para entrenar una red convolucional, entrenado una red neuronal convolucional a partir las muestras con los hiperparámetros seleccionados, descubierto que en este caso la normalización del lote influye dramáticamente en los resultados y visto cómo aumenta la velocidad de resolución de este problema concreto al resolverlo con una red neuronal. Tanto las mejores redes entrenadas como las muestras usadas y la principal parte del código puede encontrarse en: https://github.com/martinbombin/tfg.

Palabras clave: red neuronal convolucional (CNN), tensión de Von Misses, aprendizaje profundo (DL), aprendizaje automático (ML).

Tabla de contenidos

Tabla de figuras

1. Introducción

Estamos viviendo una época en la que la inteligencia artificial destaca más que nunca y donde los rápidos avances en el aprendizaje profundo hacen que ésta crezca rápidamente. De entre todos los modelos de aprendizaje profundo, las redes neuronales convolucionales han demostrado ser las que mejor rendimiento tienen para tareas de visión por computador, tales como la clasificación de imágenes (determinar cuál es el objeto principal en una imagen) [10] y detección de objetos (etiquetar los diferentes objetos que aparecen en una imagen) [11] entre otras aplicaciones.

Existen programas muy útiles para el modelado de objetos que nos permiten definir las características de éste, de su entorno y los distintos valores que se desean calcular. Estos programas se utilizan mucho en diferentes ingenierías y se basan en ecuaciones bastante costosas que calculan la solución más precisa que se puede obtener. Esto hace que lanzar una gran cantidad de instancias requiera mucho tiempo de cálculo.

Debido a los grandes avances que se están consiguiendo con las redes convolucionales, muchos grupos de investigación se plantean emplearlas en problemas ya resueltos para aumentar la velocidad a la que estos se resuelven, adaptando incluso el formato de las propiedades de los problemas (fuerzas, formas, sujeciones…) al formato de entrada de estas redes (imágenes), tal y como se ha hecho en este trabajo.

En este trabajo nos hemos enfocado en un problema concreto con unas restricciones concretas: el cálculo de la tensión de Von Misses para estructuras en voladizo en 2 dimensiones, y veremos cómo podemos mejorar la velocidad con precisiones muy similares a las obtenidas con modelos analíticos.

1.1. Motivación

Debido al auge de la inteligencia artificial en los últimos tiempos, a que la rama de la carrera que escogí (computación) está estrechamente relacionada con ésta y a que profesionalmente quiero dedicarme a este mundo en un futuro, busqué algún profesor que ofertara trabajos de fin de grado relacionados con este ámbito. Así empezamos este trabajo de final de grado donde debía aprender sobre contenidos no vistos durante la carrera para después poner en práctica este conocimiento con un problema ya resuelto mediante métodos tradicionales. Para todo ello se tomó como base un artículo que trata de resolver este mismo problema [5].

1.2. Objetivos

El principal objetivo de este trabajo es analizar la posibilidad de utilizar redes neuronales convolucionales en el cálculo de elementos finitos, analizando sus ventajas y desventajas, conseguir buenos resultados con respecto a los mejores resultados actuales que emplean otros métodos y aumentar la velocidad de cómputo con respecto a las soluciones actuales.

1.3. Estructura

En un principio hablaremos de cómo es el estado del arte del campo de estudio del trabajo (2) y se propondrá una solución al problema a resolver en base a éste (2.1.). A continuación, se explicará cuáles son los fundamentos tecnológicos en los que se basa el trabajo y como se ha llegado al punto en el que se está actualmente (3). Tras esto se expondrá la realización del trabajo, partiendo por cómo se ha implementado la red (5.1.), seguido de cómo se extrajeron las muestras de entrenamiento para poder entrenar a ésta (5.2.) y finalizando con el entrenamiento de la red neuronal convolucional y la exploración de algunos hiperparámetros (5.3.). Seguidamente se verán los resultados que hemos obtenido (6) y las conclusiones del trabajo (7). Finalmente se propondrán posibles mejoras y formas de seguir ampliando el proyecto (8).

2. Estado del arte

En este proyecto usaremos una red convolucional con una estructura codificador-decodificador para tomar como entrada una imagen con diferentes canales y transformarla en un mapa de colores de mismo ancho y alto que la imagen tomada como entrada.

Actualmente uno de los principales campos en los que se utiliza la estructura codificador-decodificador, tomando imágenes como entrada cuya salida es la imagen original transformada, es para crear mapas de segmentación [9]. Esto lo que hace es crear, dada una imagen, un mapa de colores de los diferentes elementos que pueden verse en esa imagen. Un ejemplo de esto puede verse en la Ilustración 1. Los mapas de segmentación tienen muchas utilidades, están presentes en los sistemas de conducción autónoma, en algunos sistemas de diagnóstico médico, cámaras de seguridad, etc.

Ilustración 1: Ejemplos de entradas y salidas para un problema de mapas de segmentación.

Tanto SegNet [2] como Deconvnet [3] son modelos con muy buenos resultados en estos campos que siguen este esquema codificador-decodificador.

La arquitectura codificador-decodificador también puede ser utilizada para otras tareas más simples como puede ser el reconocimiento de puntos claves de una imagen. Un ejemplo de esto último sería el reconocimiento de puntos clave en el rostro de una persona (nariz, boca, ojos…) [4].

2.1. Propuesta

Se partirá de lo realizado en [5] como base, utilizaremos la misma arquitectura para crear un modelo con Keras[1] que pueda, a partir un conjunto de muestras de entrenamiento y de prueba, calcular la tensión de Von Misses de una estructura en voladizo en 2 dimensiones.

También automatizaremos, en la medida de lo posible, la generación de muestras de entrenamiento y de test para poder generar nuestro propio conjunto de muestras con las que entrenar la red.

[1] https://keras.io/

3. Fundamentos teóricos

3.1. Redes neuronales

Las redes neuronales son un subconjunto del aprendizaje automático y el pilar principal del aprendizaje profundo. Tanto su nombre como su estructura se inspiran en el cerebro humano y en como las neuronas de éste se comunican entre sí.

Las redes neuronales están compuestas por una capa de entrada, una o más (en el caso de las profundas) capas intermedias y una capa de salida. Cada capa consta de un número de nodos que están conectados con los nodos de la siguiente capa gracias a un conjunto de pesos. Por ejemplo, en el caso de redes completamente conectadas como la de la Ilustración 2, cada nodo está conectado a todos los nodos de la capa siguiente por un peso distinto. El valor de un nodo viene dado por la suma del producto de los pesos que van a ese nodo por la entrada a la que están conectados. Al resultado anterior se le aplica una función no lineal para que el conjunto de la red no se acabe comportando como una función lineal.

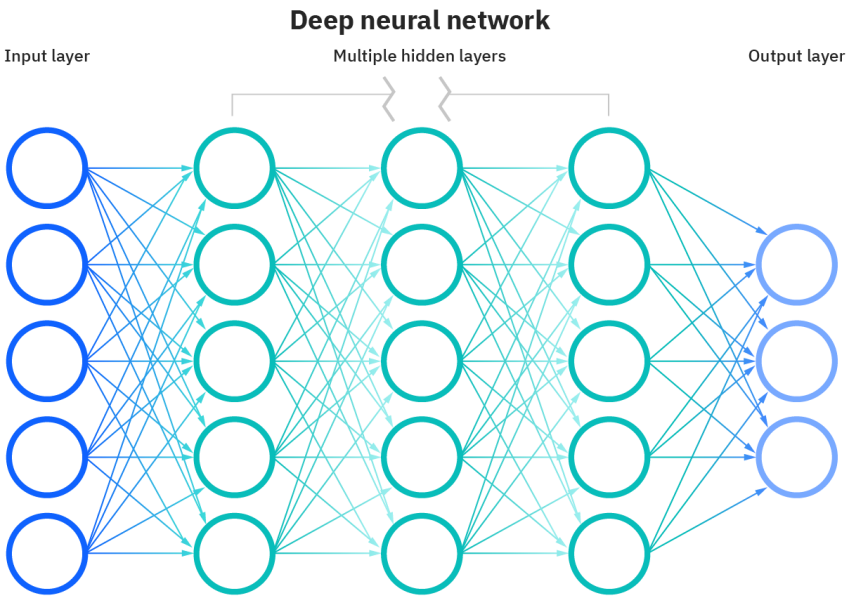

Ilustración 2: Ejemplo de red neuronal totalmente conectada con varias capas ocultas.

La dimensión de la entrada de la red coincide con los nodos de la primera capa y la dimensión de la salida con los nodos de la última capa. Para que estas redes funcionen correctamente necesitan ser entrenadas, lo que implica ir modificando los pesos de la red para obtener los resultados deseados. Se parte de un conjunto de muestras de entrenamiento, compuestas por datos de entrada y la salida que se debería conseguir con estos. Para el aprendizaje es necesario

el empleo de una función de coste, la cual comprueba como de parecido es el resultado al resultado esperado. En base al valor de la función de coste (el objetivo siempre es minimizar ésta), se utiliza un optimizador para actualizar los pesos de la red en base a los resultados obtenidos por la función de coste con una muestra concreta.

Una función de coste se utiliza para saber cómo de mala es la aproximación que ha hecho la red con respecto al valor que se esperaba como resultado. Dependiendo de la función de coste un mismo resultado puede ser mejor o peor que otro por lo que es importante elegir bien ésta. Una función de coste ampliamente utilizada y conocida es el error cuadrático medio, donde se mide el promedio de los errores al cuadrado.

Probablemente el optimizador más conocido, debido a su historia, sea el que aplica descenso por el gradiente. Éste se basa en buscar el mínimo en la función de coste aplicando funciones diferenciales, dando pasos en la dirección contraria al gradiente descendiendo así por este, obteniendo valores cada vez más pequeños de la función de coste. El tamaño de los pasos depende de un hiperparámetro llamado ratio de aprendizaje y del valor del gradiente. A mayor ratio de aprendizaje mayor será el tamaño de los pasos que se dan. Hay diversas formas de actualizar los pesos de la red, pueden actualizarse después de haber procesado todas las muestras de entrenamiento (*batch gradient descent*), en cada muestra de entrenamiento (*Stochastic gradient descent* (SGD)), o una mezcla de ambos, donde se actualizan los pesos cada vez que se procesan un número determinado de muestras (*mini-batch gradient descent*).

Una de las principales ventajas de SGD es la facilidad que tiene para saltar entre mínimos locales, pudiendo encontrar mejores mínimos por los que seguir descendiendo. El inconveniente de esto es que también el error fluctúa mucho más, dificultando a veces la convergencia. Esto ocurre especialmente en puntos muy cercanos a los mínimos locales donde el error tiende a tomar forma de elipse y el SGD empieza a zigzaguear hacia el mínimo como puede verse en la Ilustración 3 (a). Para solventar este problema se diseñó el SGD con momentum, donde se aplica SGD pero el resultado viene dado esta vez por una fracción de los resultados calculados anteriormente por el SGD sumada a una fracción de los calculados para ese paso. Su uso implica la instanciación de un nuevo hiperparámetro que indica el peso que tienen los anteriores pasos con respecto al nuevo valor calculado. Esto acelera el descenso en estos casos, como puede apreciarse en la Ilustración 3.

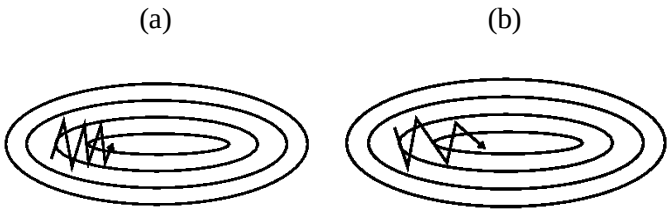

Ilustración 3: Error que forma elipsis cerca de un mínimo local. (a) Aplicando SGD; (b) aplicando SGD con momentum

Uno de los optimizadores más populares hoy en día es Adam [16] (*Adaptive Moment Estimation*), optimizador que calcula ratios de aprendizajes adaptativos para cada parámetro de la red y también aplica un mecanismo similar al momentum explicado anteriormente.

A la hora de entrenar una red, lo más sencillo y utilizado es partir de una red ya entrenada por gente con más recursos que nosotros. Pudiendo partir de una red pre-entrenada por una gran corporación que dispone de muchísimas más muestras de entrenamiento y recursos de cómputo que nosotros permite acelerar el aprendizaje de la misma adaptándola a las necesidades concretas del problema a resolver. A esta práctica se le llama transferencia del aprendizaje, ya que se intenta transferir el conocimiento adquirido en una red previamente entrenada a otro propósito. Cuanta más similitud haya entre los problemas a resolver mejor funcionará nuestra nueva red.

Para transferir el aprendizaje de una red, lo más común es partir de la red entrenada y congelar las primeras capas de esta antes de realizar el entrenamiento. Esto provoca que no se actualicen los pesos de esas capas durante el entrenamiento, mantener los fijos. La ventaja es que las capas iniciales adquieren conocimiento de bajo nivel que puede ser útil en el entrenamiento de las capas finales que es donde se lleva a cabo la abstracción a más alto nivel. El número de capas a congelar dependerá del número de muestras que tengamos, cuantas más capas congelemos menor flexibilidad tendremos para aprender nuevas características. Éste será pues el camino si disponemos de pocas muestras, puesto que tampoco se iba a poder aprender muchas características igualmente. En cambio, si tenemos muchas muestras nos interesará congelar solo unas pocas capas. Una restricción del transfer-learning es que sólo lo podremos hacer cuando partamos de una red con la misma estructura que la que queremos conseguir. Si la estructura de la red de la que partimos no nos convence podemos congelar la red entera y simplemente añadirle nuevas capas al final, de nuevo en función del número de muestras que dispongamos. De esta forma, las capas a entrenar se usarán para convertir las características de más bajo nivel aprendidas por la red de la que partimos en un resultado que se adapte a nuestras muestras.

3.2. Redes neuronales convolucionales

Las redes neuronales convolucionales se basan en la forma en la que los humanos percibimos nuestro alrededor a través de la visión. En ésta se distingue entre neuronas simples y neuronas complejas en la corteza visual primaria, empezando así el proceso de visualizar por estructuras sencillas como pueden ser los bordes [12]. Esto da lugar a una jerarquización en el proceso de visualización que se acabó implantando en forma de redes neuronales convolucionales [8]. En éstas, las primeras capas se excitan con formas muy simples como bordes, líneas o esquinas[2] y las últimas con patrones más complejos como pueden ser ojos, narices o incluso cabezas de algunos animales[3].

[2] https://microscope.openai.com/models/alexnet/conv1_1_0?models.op.feature_vis.type=channel&models.op.technique=feature_vis

[3] https://microscope.openai.com/models/alexnet/Relu_0?models.op.feature_vis.type=channel&models.op.technique=feature_vis

El origen de este tipo de redes tiene lugar en 1988, donde, inspirado por lo visto en [13], Kunihiko Fukushima creó el Neocognitron. Ésta era una red neuronal con varias capas que podía reconocer imágenes no viéndose afectada por desplazamientos en la imagen. Tras esto, Yann LeCun llevó la teoría a la práctica y creó LeNet-5, la cual fue la primera red convolucional desarrollada para el reconocimiento de dígitos manuscritos [14].

Este tipo de redes no empezó a popularizarse hasta 2012, gracias a los buenos resultados de un modelo de este tipo (un orden de magnitud mejor que el anterior mejor modelo de su época) [10] que usó tarjetas gráficas para el entrenamiento de su red. Esto último permitió aumentar la velocidad de entrenamiento de la red de forma vertiginosa y permitió entrenarla más. El entrenamiento de redes convolucionales con tarjetas gráficas hoy en día es básico. Esto fue el inicio de una rápida evolución en el campo de la visión por computador.

Las redes neuronales convolucionales se basan, como bien indica su nombre, en las convoluciones. Éstas consisten en tomar una imagen de entrada y aplicarle un filtro (normalmente bastante más pequeño que la imagen de entrada) que se va desplazando por la imagen. El tamaño del filtro en altura y anchura se especifica de antemano, pero la profundidad de éste vendrá siempre dada por la profundidad de la entrada que reciba. El filtro recorre los valores de la entrada con desplazamiento horizontales y verticales y genera una salida sumando el producto de los valores del filtro que coincidan con los valores de la entrada en cada desplazamiento. El tamaño de los desplazamientos puede variar, tanto en vertical como en horizontal. Los valores de los filtros son los principales pesos que ha de aprender en una red neuronal convolucional. Un ejemplo de imagen y filtro puede verse en la Ilustración 4.

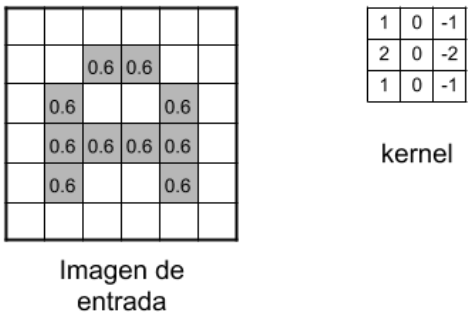

Ilustración 4: Ejemplo de entrada y filtro.

Se puede añadir relleno a la imagen de entrada antes de realizar la convolución. Por ejemplo, un tipo de relleno llamado *same* añade ceros alrededor de la entrada, aumentando el alto y ancho de la entrada lo necesario para que tras hacer la convolución el resultado tenga las mismas dimensiones que la entrada. Este tipo de relleno se utiliza para no perder información de las esquinas, puesto que si no se empleara relleno, las esquinas solo afectarían a un valor de la salida (debido a que, si no, el kernel se saldría de la imagen) mientras que aplicando este relleno las esquinas influencian lo mismo en el resultado que los puntos que se encuentran en mitad de

la imagen de entrada. Un ejemplo de una convolución con la imagen y filtros de la Ilustración 4 y su resultado sin aplicar relleno puede verse en la Ilustración 5.

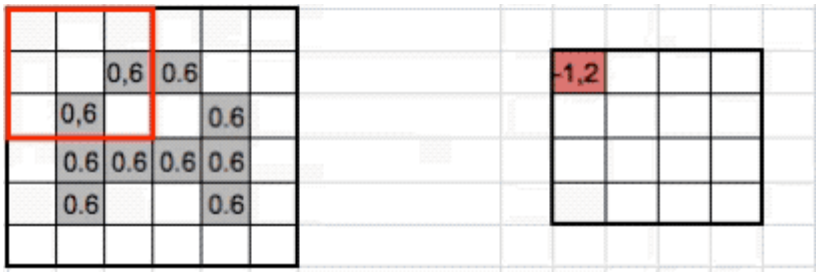

Ilustración 5: Ejemplo de convolución de la entrada y el filtro de la ilustración anterior. En este caso el filtro se desplaza 1 posición en horizontal, una en vertical cada vez y no se aplica relleno.

Al crear una capa convolucional para una red se suele definir el número de filtros que se aplicarán en esa capa, el tamaño de estos, el tamaño de los desplazamientos que harán, el relleno empleado y, solo sí se trata de la primera capa de la red, se especificará también el tamaño de la entrada que recibirá esa capa. Las dimensiones del resultado dependerán de todos los parámetros escogidos, siendo el número de filtros equivalente a la profundidad de la salida de esa capa.

Normalmente tras una capa convolucional se añade una función de activación (al igual que se hacía con la redes convolucionales convencionales) para introducir no linealidad a la función y que ésta pueda aprender características complejas. La más usada actualmente es la *Rectified Linear Unit* o ReLU, que consiste en mantener el valor de la entrada si éste es mayor que 0 o truncarlo a 0 si es menor.

En 2015, un nuevo modelo llamado res-net [6] consiguió los mejores resultados hasta el momento, con solo un 3,57% de error en las 5 predicciones más probables para cada entrada en ImageNet. ImageNet es un conjunto de imágenes muy usado para probar el funcionamiento de modelos relacionados con imágenes. Los resultados obtenidos demostraron que Res-Net era capaz de funcionar mejor que incluso los validadores humanos [1], como puede verse en la Ilustración 6.

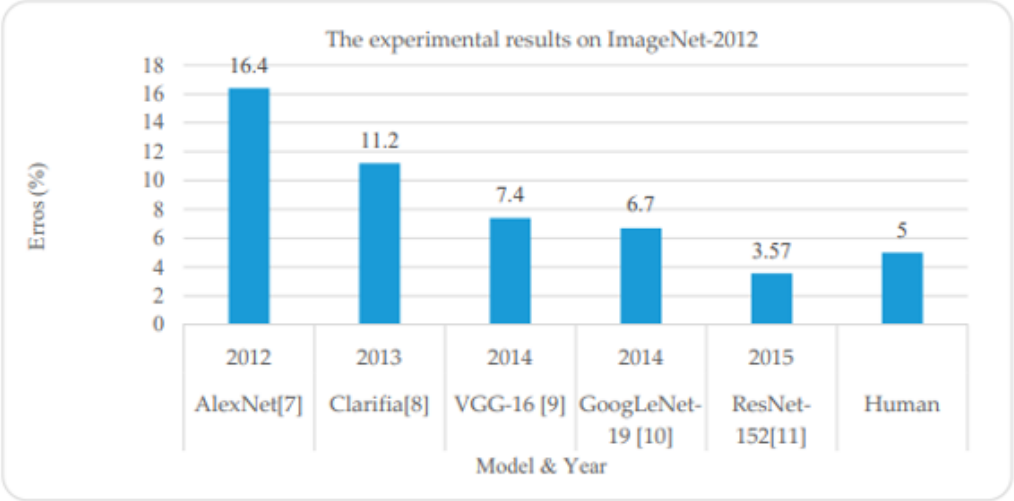

Ilustración 6: Precisión para el conjunto de datos ImageNet con diferentes modelos de Deep Learning.

La innovación introducida por este modelo fue el uso de un nuevo tipo de capa llamada capa residual. Esta nueva capa consiste en dos capas convolucionales normales consecutivas cuya entrada es sumada a su salida, tal y como puede verse en la Ilustración 7. Esto facilitó el aumento de la profundidad de estas redes, puesto que ahora la red podía decidir si era necesario aumentar la complejidad del modelo ajustando los pesos de una capa concreta o simplemente mantener una matriz de ceros en esa capa y mantener el valor que tomaba como entrada de la capa anterior. Antes de esto a una capa concreta le costaba mucho trabajo aprender la función identidad para simular este funcionamiento. Así, la introducción de las capas residuales permitió mantener las ventajas que ofrece una red poco profunda y una red profunda al mismo tiempo, pudiendo adaptarse rápidamente a funciones más complejas debido a su profundidad o a más simples ignorando capas innecesarias.

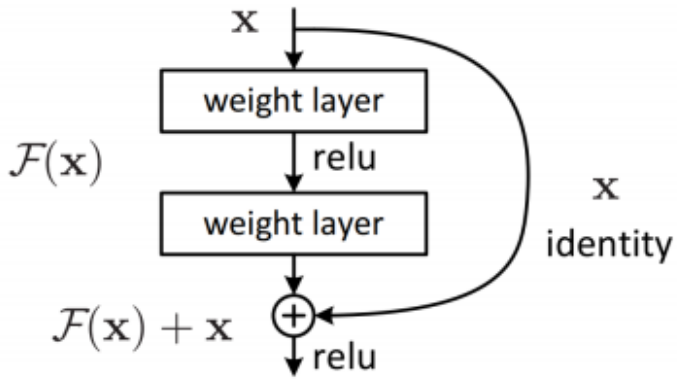

Ilustración 7: Capa residual: bloque con un atajo.

También, en 2015 surgió, el *Batch normalization* [15]. Normalmente las redes neuronales se entrenan por lotes, donde todas las muestras del lote son introducidas en la red en paralelo y se

actualizan los pesos de la red una vez esta ha terminado con todo el lote. Esto sirve principalmente para reducir el tiempo de cómputo. *Batch normalization* actualmente se utiliza en la mayoría de las redes y se emplea como una capa adicional que normaliza la entrada en ese punto a través de todo el lote, suele utilizarse antes o después de las capas de activación. Normalizar de esta manera los lotes consigue que el entrenamiento sea más rápido y que puedan usarse ratios de convergencia mayores sin sacrificar una buena convergencia como puede verse en la Ilustración 8 sacada de [15].

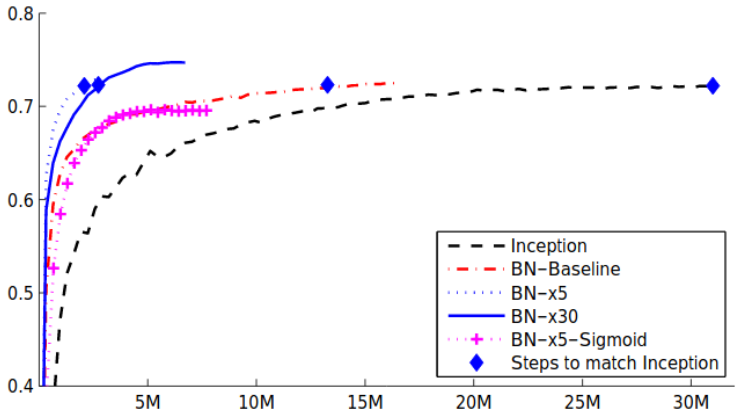

Ilustración 8: Impacto de *Batch normalization* en el entrenamiento, donde BN-baseline es el modelo Inception pero con *Batch normalization*, x5 indica que es el mismo modelo (Inception) pero con un ratio de aprendizaje mayor y *Sigmoid* indica que se sustituyó la función de activación (ReLU) por una de tipo *Sigmoid*.

Otro avance menos popular que los anteriores pero también ampliamente utilizado es el que se explica en [7]. Éste consiste en una nueva capa, llamada por ellos *Squeeze-and-Excitation* o SE, que se encarga de ponderar los diferentes canales de la imagen dependiendo de la importancia que tengan para el computo. Anteriormente todos los canales de la imagen ponderaban lo mismo.

Así llegamos al artículo de referencia [5], donde se combinan ambos tipos de capas para crear la capa mostrada en la Ilustración 9. Esta propuesta consta de 2 capas convolucionales seguidas de una capa SE donde incluyen un atajo desde la entrada de la primera capa hasta la salida de la capa SE. Éste será el principal bloque de la arquitectura de nuestra red.

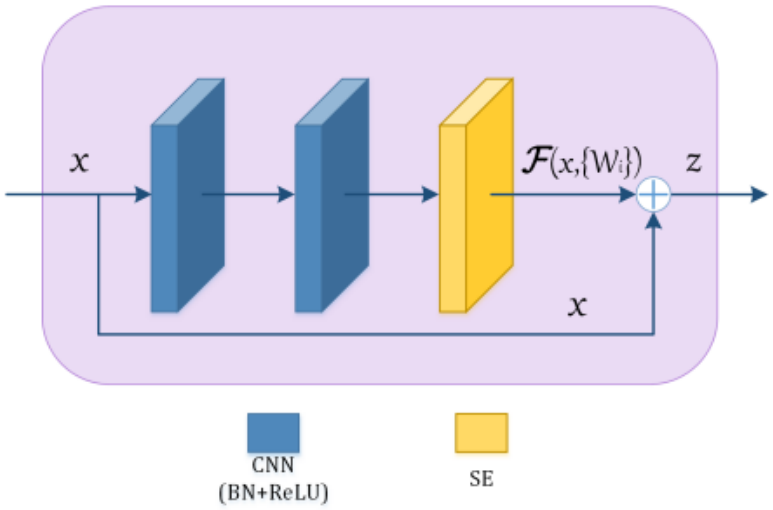

Ilustración 9: Bloque SE-Res: combinación de capa residual con capa SE.

Así, nuestra red estará compuesta por 3 capas convolucionales que se encargaran de reducir la dimensionales de nuestra imagen de entrada, 32 píxeles de ancho x 24 píxeles de alto con 5 canales, aumentando el número de canales a su vez. La zona central constará de 5 capas como las de la Ilustración 9 que mantendrán las dimensiones de la entrada y 3 capas convolucionales que se encargaran de devolverle las dimensiones iniciales a la última del tipo de la Ilustración 9 pero esta vez con un solo canal y no 5. De este modo se crea una estructura codificador-decodificador [8] donde la entrada se va codificando a medida que atraviesa la red y, a partir de la representación intermedia obtenida en las capas intermedias, después se decodifica para formar el resultado final. Tras cada capa convolucional se aplicará ReLU como función de activación y tras esto se aplicará también *Batch normalization*. Puede verse esta arquitectura y un ejemplo de su entrada y salida en la Ilustración 10.

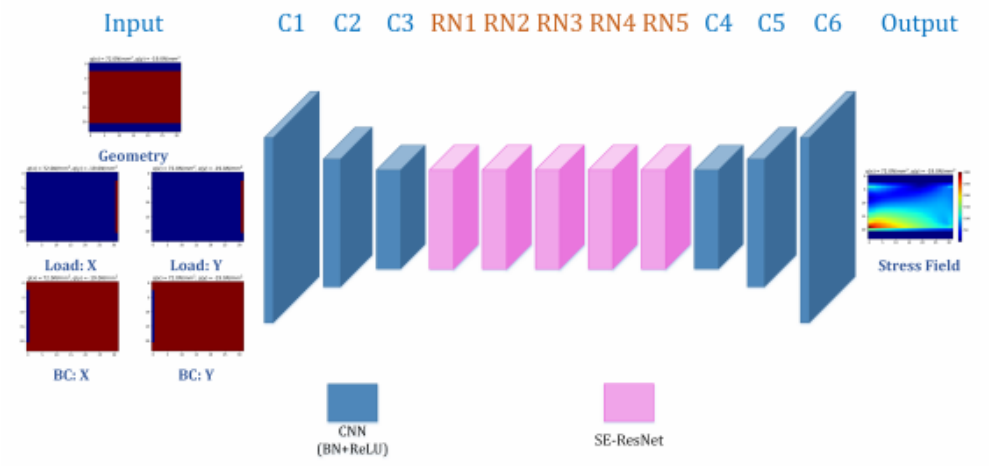

Ilustración 10: Arquitectura a utilizar con una posible entrada y su salida.

4. Metodología

El trabajo puede dividirse en 2 bloques principales, siendo el primero la generación de muestras de test y de entrenamiento y el otro la creación del modelo y su entrenamiento.

A la hora de crear las muestras de entrenamiento se creará un *script* para cada forma diferente con la que se quiera entrenar el modelo para, a raíz de este, poder automatizar el extracto de múltiples instancias con diferentes fuerzas y magnitudes aplicadas. Una vez conseguidas las muestras se ha podido entrenar el modelo y se ha probado su funcionamiento.

Aunque parezca que el proceso es secuencial, gracias a que las muestras usadas en [5] están subidas en el git-hub que referenciaban, se ha optado por ir haciendo ambas partes de forma paralela. De este modo, se han podido cambiar las muestras del artículo por las nuestras una vez se ha comprobado que la red funcionaba correctamente y estaba bien construida.

4.1. Tecnología utilizada

La red neuronal se ha programado en Python con la API especializada en inteligencia artificial llamada Keras. Para facilitar la instalación y la importación de librerías se ha utilizado Colab[4], que ya disponía de todas las librerías necesarias instaladas y, de forma gratuita, pero con restricciones, ofrecía 25GB de memoria RAM (que más tarde acabó recortando a solo 12) y una GPU con la que podemos lanzar nuestro código de forma remota (así como sincronizarlo con una cuenta de Google para usar Drive).

Para la extracción de muestras se ha utilizado Ansys[5], un simulador usado en diferentes ingenierías que nos permite obtener, entre otras cosas, la tensión de Von Misses de cada punto de un objeto con las características físicas y geométricas, fuerzas aplicadas y sujeciones que previamente hayamos definido. También usaremos Matlab para facilitar la automatización de las múltiples ejecuciones de Ansys.

La tensión de Von Misses es ampliamente utilizada como indicador de la resistencia de una estructura a las fuerzas a las que está sometida. De este modo, se puede saber si la estructura aguantará o no la tensión[6] a la que va a ser sometida.

También utilizaremos las siguientes librerías Python para facilitar el trabajo:

- SciPy[7], una libreira de código abierto para matemáticas, ciencias e ingenierías que usaremos para tratar con las muestras conseguidas mediante Matlab (ficheros .mat),
- NumPy[8] para facilitar el manejo de vectores y matrices,

[4] https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index
[5] https://www.ansys.com/
[6] https://www.urbipedia.org/hoja/Tensi%C3%B3n_de_Von_Mises
[7] https://www.scipy.org/
[8] https://numpy.org/

- Matplotlib[9] para representar los datos como imágenes en 2 dimensiones,
- Pickle[10] para almacenar otros tipos de datos en disco o para cargarlos en memoria y
- ipython-autotime[11] para medir los tiempos de ejecución en Python.

[9] https://matplotlib.org/
[10] https://docs.python.org/3/library/pickle.html
[11] https://pypi.org/project/ipython-autotime/

5. Desarrollo

Como se dijo anteriormente el desarrollo de la solución propuesta puede dividirse en 2 bloques principales: la extracción de muestras de entramiento y la implementación de la red. Mantendremos el orden cronológico, ya que se empezó por la implementación, posteriormente se llevó a cabo el entrenamiento de la red con muestras sacadas del artículo de [5] y, finalmente, se realizaron un conjunto de experimentos con los que decidir el valor de algunos hiperparámetros y la forma de entrenar la red final.

5.1. Implementación de la red

Antes de poder siquiera empezar con la implementación, hemos tenido que documentarnos al respecto. El punto de partida en el que se encuentra el alumno es haber terminado las asignaturas de la rama de computación, donde se llegan a introducir las redes neuronales totalmente conectadas poco profundas y sus fundamentos, y haber realizado por cuenta ajena unos cursos introductorios de edX[12]

Por lo tanto, para poder empezar, el alumno hizo el siguiente curso de Coursera[13] donde se explica con mayor profundidad, desde lo más básico a otros temas más complejos, las redes neuronales convolucionales y como se implementan. También tuvo que estudiar en profundidad el artículo de referencia [5] y otros artículos referenciados por éste, como pueden ser [6], [7] y [8].

Tras esto, se empezó la implementación de nuestra red. El tamaño de la imagen original, de las entradas y salidas de cada capa y la arquitectura de la red es el mismo utilizado en [5], recibiendo como entrada imágenes de 32 x 24 píxeles con 5 canales:

- El primero siendo una matriz de unos y ceros que dibujan la figura a la que se le aplicará la fuerza, donde un uno implica que ese pixel pertenece a la figura y un 0 que no,
- El segundo la magnitud de la fuerza homogénea en el eje X que se aplica y donde se aplica,
- El tercero es lo mismo pero para el eje Y,
- El cuarto es la sujeción en el eje X y donde se encuentra esta, siendo una matriz de 0 y -1 valores ,donde -1 implica que hay una sujeción en ese pixel, y
- El quinto es lo mismo que la anterior pero para el eje Y.

Un ejemplo de lo anterior y la estructura de la red puede verse en la Ilustración 10.

[12] https://www.edx.org/professional-certificate/deep-learning?index=product&queryID=ebd9d77d5b2190d96e6fa8ab8a7018e6&position=1

[13] https://www.coursera.org/learn/convolutional-neural-networks

A la hora de implementarlo se creó una clase Python para formar el bloque SE-Res visto en la Ilustración 9. Esta clase tiene un constructor donde se definen 2 capas idénticas, formadas por una capa convolucional con 128 filtros, tamaño de estos de 3x3 y desplazamientos de 1 tanto para el ancho como para el alto seguida de una normalización del lote. Tras estas dos capas viene la capa SE explicada en [7]. Después se define la función que se llamará cuando se quiera usar este bloque. Esta función simplemente hará pasar la entrada que reciba por todas las capas formando una tubería y devolverá el resultado de la última capa sumado a la entrada original, formando así el atajo.

Después creamos el modelo compuesto por 3 capas convoluciones cuyas funciones de activación son RELU y que, tras la función de activación, se aplica una normalización del lote. Todas utilizan el mismo relleno, *same*. La primera recibe como entrada un tensor de 3 dimensiones de 24 x 32 x 5 y tiene 32 filtros con un tamaño de 9 x 9, la segunda tiene 64 filtros de 3 x 3 y la última 128 de 3 x 3. Las dos primeras tienen desplazamientos de 2 para el alto y el ancho mientras que la tercera lo tiene de 1 para ambos. Tras estas 3 capas vienen 5 del tipo SE-Res definidas en el párrafo anterior. Finalmente añadimos las 3 capas convolucionales transpuestas, todas con activaciones RELU y relleno *same*, las dos primeras con tamaños de filtros de 3 x 3 y la última con tamaño de filtro de 9 x 9, se aplica normalización del lote tras las 2 primeras, la primera capa costa de 64 filtros, la segunda de 32 y la última de 1. Se escoge el optimizador Adam con sus valores predeterminados, como función de coste el error cuadrático medio y en principio a la hora de entrenar se usará un tamaño de lote de 256. Todas estas características se han sacado de [5].

Una vez construido el modelo con Keras en Colab, se han subido al Drive de la cuenta asociada a Colab las muestras de [5] para poder así usarlas para entrenamiento. Estás muestras estaban en un archivo .np, que es la extensión usada para almacenar objetos de Numpy, lo que acelera y facilita su uso.

Las muestras vienen en un formato concreto que se ha tenido que descubrir haciendo diversas pruebas. Al cargar las imágenes en un principio se consigue una estructura bidimensional donde, por el tamaño de estas, se deduce que la primera dimensión es el número de muestras y las segunda equivale al ancho por el alto por el número de canales. Esto era un inconveniente puesto que la entrada de nuestra red es un tensor tridimensional, no un vector. Por lo que ha habido que cambiar la estructura de los datos para poder utilizarlos en la red. Tras mostrar por pantalla diversas agrupaciones de 32 x 24 píxeles de la segunda dimensión nos dimos cuenta de que los valores estaban ordenados primero por fila, luego por columna y finalmente por canales, teniendo que mostrar los primeros 24 valores como la primera fila, los 24 siguientes como la segunda, etc. Con esta información se procedió a reestructurar los datos.

Utilizaremos 2 tipos distintos de normalización para comprobar cual funciona mejor:

- Tipo 1: restar la media y dividir entre la desviación típica, restar el valor mínimo y
- Tipo 2: dividir entre el valor máximo menos el mínimo (normalización min-max).

También se probará a no aplicar ningún tipo de normalización.

Para todo lo anterior se ha creado una función que inicializa los datos de [5] de 3 formas diferentes dependiendo de lo que se le indique. Esta función crea los distintos conjuntos de entrenamiento y test, separa las entradas de las salidas, baraja las muestras con una semilla determinada, cambia la estructura de la entrada para que tenga el formato que necesita la red

como entrada y, mediante operaciones matriciales, normaliza los datos de entrenamiento de las 2 formas comentadas anteriormente o no, dependiendo de lo que se le haya indicado. Esta función devuelve los datos listos para entrenar la red y las medias/mínimos y deviaciones típicas/máximos de cada canal para poder normalizar futuras muestras (entre ellas las muestras de test) devolviendo una lista vacía en el caso que se escogiera no normalizar.

Una vez hecho todo lo anterior, se entrena la red durante 1000 (el máximo número de iteraciones que haremos con 120000 muestras será de 1000 en lugar de las 5000 que hicieron en [5] debido a limitaciones temporales y de hardware) tomando el 80% del conjunto de muestras como muestras de entrenamiento y el 20% restante como muestras de test (que normalmente son los ratios que se utilizan) que serán las que se utilizarán para validar los resultados.

El error cuadrático medio para cada tipo de normalización es: 2096 para el primero (está lejos de funcionar correctamente), 0,4714 para la normalización min-max y 0,3363 para las muestras sin normalizar. Viendo que lo que mejor funciona en este caso es no normalizar las muestras, de aquí en adelante los experimentos se harán de esta forma. En la Ilustración 11 puede verse los resultados de una muestra aleatoria del conjunto de test con los diferentes tipos de normalización y como es imperceptible la diferencia entre la min-max y la que no se normaliza.

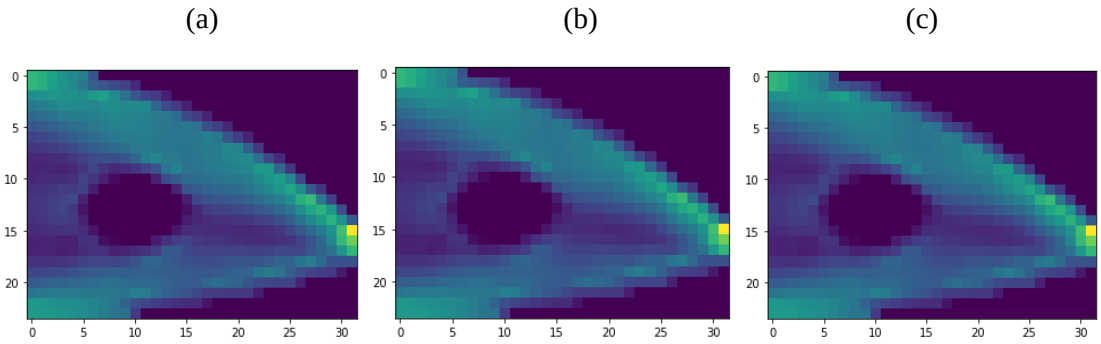

Ilustración 11: Resultado esperado y resultados con normalización min-max y sin normalizar. (a) resultado esperado; (b) resultado con min-max; (c) resultado sin normalizar.

En la Ilustración 12 puede verse como converge el error y el logaritmo en base 10 del error a través de las distintas iteraciones del entrenamiento de la red con las muestras sin normalizar, debido a la dificultad para visualizar la progresión en el error sin alterar, de ahora en adelante se presentarán solo gráficas del logaritmo en base 10 de éste.

(a) (b)

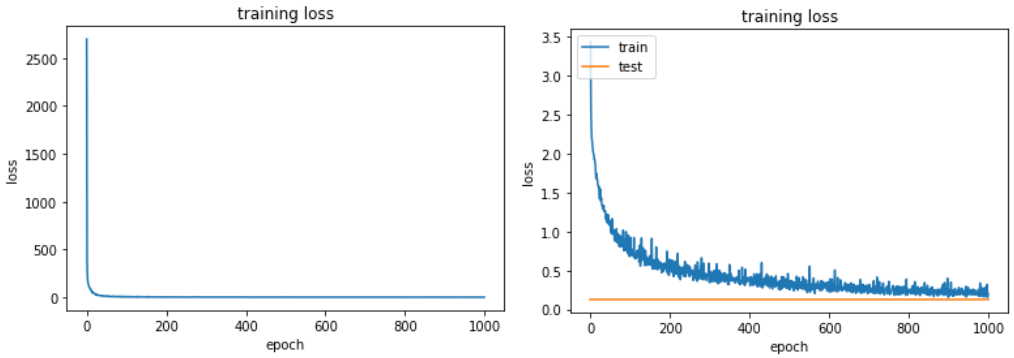

Ilustración 12: Error cuadrático medio durante el entrenamiento de la red. (a) Error sin alterar; (b) logaritmo en base 10 del error en entrenamiento y del error final sobre el conjunto de test con la red entrenada.

Este experimento se hizo cuando teníamos 25 GB de RAM. Ahora sólo disponemos de 12 por lo que no se podrían hacer 1000 iteraciones con tantas muestras en una sola ejecución. Habría que hacer menos iteraciones e ir guardando el resultado intermedio en disco para después volver a hacer más iteraciones, lo que ha complicado la realización de futuros experimentos.

5.2. Extracción de muestras de entrenamiento

Esta es la parte del proyecto que más tiempo a llevado, primero se extrajo la información relevante de [5] que indicaba cómo se habían sacado las muestras. Estamos tratando con figuras en 2 dimensiones, de un material isotrópico concreto (el mismo para todas las muestras). La figura tiene de dimensiones 1 milímetro de largo por 0,75 milímetros de ancho[14] dando lugar a una imagen de 32 x 24 píxeles. La pieza está sujeta por el lado izquierdo tanto en el eje Y como en el eje X y los elementos componentes son cuadrados formados por 4 nodos (los elementos son los puntos a los que Ansys da un valor final al calcular la tensión de Von Misses en Ansys y su valor viene dado por la interpolación del valor calculado en sus nodos que, en este caso, serían las esquinas de los elementos) y las fuerzas se aplican en intervalos de 5 newtons desde -100 hasta 100 newtons tanto en el eje X como en el eje Y (buscando todas las combinaciones posibles de fuerzas). El resultado vendrá dado en megapascales.

Tras esto se contactó con la co-tutora María José, quien una vez escuchado lo anterior, compartió con el alumno 3 prácticas de asignaturas de otras ingenierías donde se explicaba cómo crear una figura en Ansys, sujetarla y aplicarles fuerzas. Además, también explicó como una vez hecha una figura en Ansys se puede extraer el fichero con extensión .log para después poder crear esa figura sin necesidad del entorno gráfico y cómo crear figuras con agujeros. Después de que el alumno estudiara las prácticas, la profesora María José hizo un breve tutorial

[14] Realmente mide 32 milímetros de ancho x 24 milímetros de alto, el alumno confundió en su momento el tamaño del elemento (coincide con cada píxel de la figura en este caso) con el tamaño total de la figura, esto no afecta sustancialmente al problema a resolver.

para que el alumno aprendiera a usar Ansys. Así se empezaron a hacer pruebas con una figura sencilla.

Uno de los principales problemas fue la creación de figuras cuyo número de elementos coincidiera con el número de pixeles tanto de ancho como de alto ya que Ansys no está enfocado en la creación de este tipo de figuras poco realistas. En un principio, se barajó la posibilidad de interpolar los puntos deseados partiendo de una solución más exhaustiva de la necesitada. Por ejemplo, interpolar 32 x 24 elementos a partir 1000 x 750 calculados con Ansys de una figura no pixelada. Pero, finalmente, se optó por tratar de obtener de Ansys una solución ya en el formato deseado, haciendo la solución lo más precisa posible para una imagen pixelada. Esto dificultó, en principio, el generar la figura pero ahorraba tiempo en el proceso de interpolar los resultados. Esto se consiguió haciendo que el tamaño de los elementos fuera igual a 1/32, o lo que es lo mismo, 0,75/24 y a diseñar las figuras mediante conjuntos de rectángulos.

Para crear la primera figura, primero elegimos el tipo de elemento con el que íbamos a tratar. En nuestro caso el 182 de Ansys que es un tipo de elemento sólido, plano y formado por 4 nodos. Tras esto elegimos las propiedades del material: en nuestro caso elegimos las del aluminio (ya que nos es indiferente mientras sean las mismas para todas las muestras). Finalmente, además, se eligió un material isotrópico con módulo de elasticidad de 70 gigapascales y coeficiente de Poisson de 0.3.

Después pasamos a definir el mallado. En un principio se optó por mallar la figura escogiendo el número de divisiones. Esto nos mallaba la figura con un mismo número de elementos tanto de ancho como de alto, lo cuál no es lo que deseábamos. Nosotros queríamos tener más divisiones a través del ancho ya que nuestra figura es rectangular y no cuadrada. Por lo que, como se explicó en el párrafo anterior, se determinó un tamaño de mallado concreto (1/32) que hiciera encajar los elementos de la figura con el número que deseábamos y la mallamos de esta forma.

Una vez teníamos la figura mallada había que aplicar las sujeciones, por lo que se seleccionó el borde izquierdo de la figura y se fijó en ambos ejes para que esta no se desplazara. A continuación, le aplicamos las fuerzas deseadas, una presión negativa en el eje X por el lado derecho de la figura (ya que una presión positiva va en sentido contrario al eje X) y una fuerza homogénea en el eje Y para cada nodo del lateral derecho de la figura. La aplicación de la fuerza homogénea en el eje X es trivial puesto que Ansys facilita su aplicación empleando una presión. Ahora bien, a la hora de aplicar la fuerza homogénea en el eje Y ha habido que dividir la fuerza deseada por el número de nodos en los que se va aplicar y aplicar este valor a cada nodo por separado (también se puede escoger una línea de nodos para no tener que ir seleccionando uno a uno). Esto es una aproximación, ya que la fórmula para aplicar una fuerza homogénea de estas características depende de la forma de la figura, siendo la del rectángulo la que se ve en la Ilustración 13 donde la fuerza es distinta en los bordes respecto al resto de puntos (esto se debe a que la fuerza depende del número de elementos adyacentes que en el caso de los bordes es menor). Una cuestión a tener en cuenta es que cambiar la formula dependiendo de la figura complica mucho el trabajo puesto que hay figuras de las cuales no conocemos la fórmula.

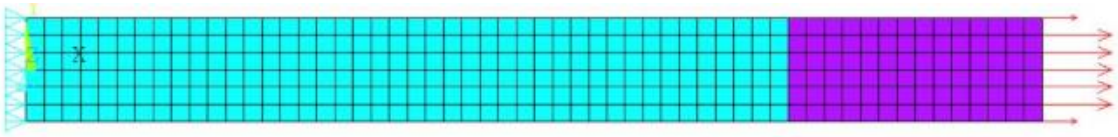

Ilustración 13: Aplicación de una fuerza homogénea en el eje X (o presión) a un rectángulo.

Una vez conseguida una primera figura, solo faltaba resolverla y mostrar, de entre todos los resultados que nos facilita Ansys, la tensión de Von Misses, (ver Ilustración 14). Como puede observarse la figura se desplaza debido a la fuerza aplicada. Puesto que el problema a resolver es el del cálculo de la tensión de Von Misses y no el como se deforma la figura, estos desplazamientos se ignorarán al trabajar con las muestras.

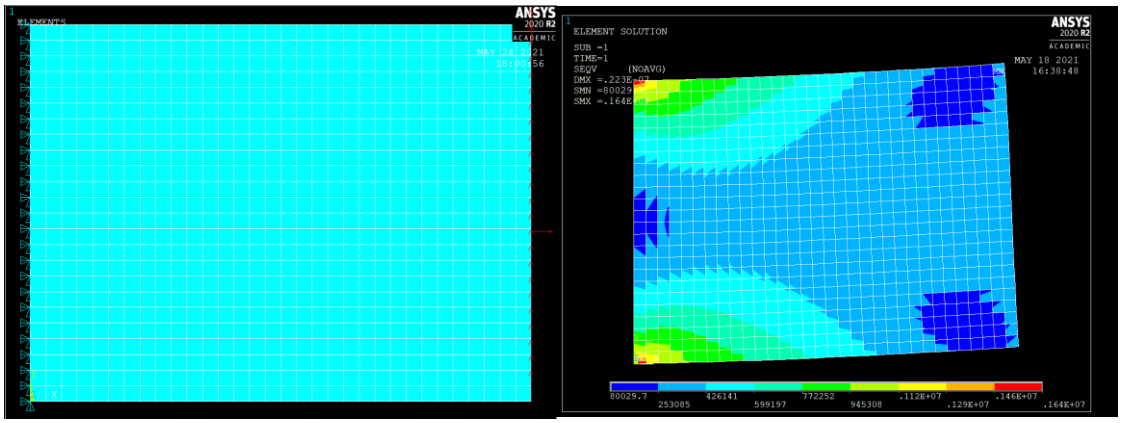

Ilustración 14: Primera pieza completa hecha en Ansys que sigue las restricciones del problema.

Una vez que se vio que hacía la figura correctamente se pasó a extraer el fichero .log y se empezó a trabajar con las figuras sin necesidad de utilizar la interfaz gráfica. Se añadieron variables al principio del fichero para poder modificar fácilmente la magnitud de las fuerzas y se colocaron estas variables en sus respectivos lugares sustituyendo las magnitudes reales de las fuerzas. Tras unas pocas pruebas el fichero .log estaba listo para ser lanzado con Matlab.

Los siguiente fue crear un fichero .m para lanzarlo en Matlab. Matlab nos permite lanzar ejecuciones de Ansys desde el mismo fichero .m, por lo que mediante Matlab podemos ir cambiando las magnitudes de las fuerzas de un fichero .log con la figura deseada, lanzar el fichero en Ansys y almacenar los resultados en una matriz que luego guardaremos en un fichero .mat.

Para ello primero necesitamos modificar el fichero .log para que guarde los resultados que necesitamos en Ansys en diferentes ficheros. Por ello, se añadieron los comando para crear una tabla con las tensiones de Von Misses de cada elemento y luego se añadió el respectivo comando para guardar la tabla en un fichero .txt. Otros datos de vital importancia que se guardaron en diferentes ficheros fueron las coordenadas X e Y de cada nodo y el identificador de los nodos que forman cada elemento.

Para empezar con Matlab se partió de un fichero proporcionado por María José donde se lanzaba un .log con Matlab en Anys y luego se pintaba el resultado por pantalla, A raíz de este

fichero, se creó un programa en Matlab que modificara un fichero .log cambiando las fuerzas en X y en Y, que creara las matrices necesarias para la entrada de la red (siendo estas forma de la figura, sujeciones y fuerzas) y que juntara estas matrices con el formato que se requiere. Finalmente, el algoritmo guardaba la matriz en un fichero .mat.

Para crear las matrices necesarias para la entrada de la red lo único que se necesitaba eran las fuerzas, generadas por el programa Matlab, y la figura. A raíz de una matriz con unos donde había figura y ceros donde no había elemento se podían sacar el resto de las matrices. Se optó por inicializar la matriz figura a mano por simplicidad, aunque se podría haber creado a partir de los resultados devueltos por Ansy ya que a partir de la posición de cada nodo y sabiendo el tamaño de cada elemento se podría calcular la posición de cada elemento y poner a 1 los valores de una matriz de ceros que coincidan con la posición de los elementos de la figura. Esta técnica es parecida a la que usamos para determinar dónde colocar cada valor de la tensión de Von Misses que nos da Ansys, puesto que nos la da ordenada por el número de elemento y no por la posición en la figura.

Cargamos en memoria las matrices que corresponden a las posiciones de los nodos, los nodos formados por un elemento y la tensión de Von Misses de cada elemento. Después recorreremos la matriz figura y cada vez que encontramos un 1 buscamos los nodos que formen ese elemento, para ello necesitamos lo que mide el lado de cada elemento, que es constante y conocido (1/32 milímetros) y los índices de la matriz figura, así sacamos la posición de los 4 nodos que forman ese elemento en la figura y buscamos los identificadores de estos nodos en la matriz correspondiente. Esto no se pudo hacer de forma exacta puesto que eran valores reales y a veces por fallos de aproximación no encajaban los valores de las posiciones, por lo que se tuvieron que usar inecuaciones donde se restaran las posiciones reales de cada nodo con el previsto de cada nodo y comprobar que el valor resultado fuera menor que el lado del elemento dividido entre 2.

Una vez que tenemos los nodos que forman un elemento hay que buscar el identificador del elemento en cuestión. Para ello debemos buscar el elemento que este formado por los 4 nodos. Esto se hizo con numerosos condicionales puesto que la posición en la aparecen los nodos en la matriz que nos da Ansys es arbitraria y hay que barajar todas las opciones posibles.

Ya que tenemos el identificador del elemento solo hay que coger su resultado del vector correspondiente y almacenarlo en una matriz en la posición del elemento que habíamos leído de la figura. Pudiendo hacer esto para cada elemento de la figura.

Así podíamos, para cada figura, ir alterando las fuerzas dentro de un rango y conseguir la tensión de Von Misses en formato matricial y coincidiendo con el formato que queremos para las muestras de entrenamiento.

Una vez se consiguió que todo esto funcionara ya teníamos una forma de crear todas las muestras necesarias a raíz de una figura. En ese momento nos dimos cuenta de que nuestros resultado no coincidían con los resultados que se obtenían en las muestras de [5]. Tal y como puede verse en la Ilustración 15, las tensiones de las muestras del artículo son muy altas en algunos puntos que no tienen sentido, tras hablarlo con María José, que era la experta en este ámbito, y después de que comprobara el artículo ella misma, llegamos a la conclusión de que habían restricciones del problema que no venían detalladas en el artículo. Eso nos forzó a solo poder utilizar nuestras muestras a partir de ahora para entrenar la red, puesto que el problema a resolver ya no era el mismo.

(a) (b)

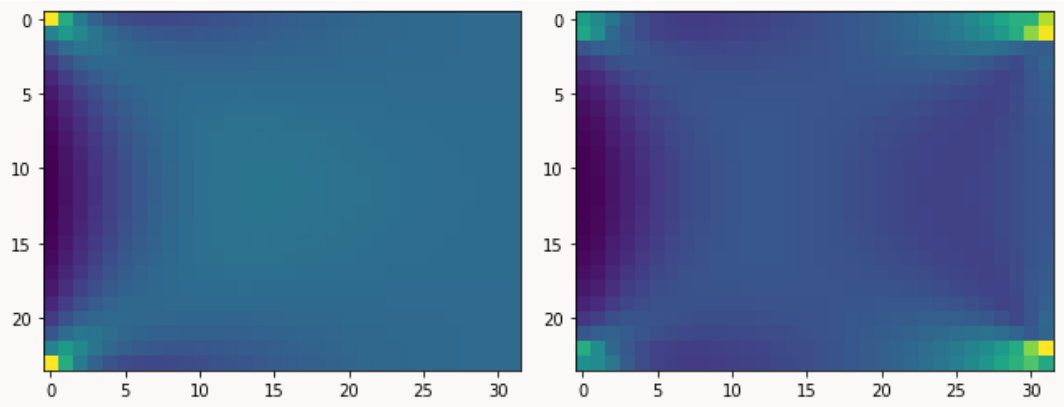

Ilustración 15: Comparación de resultados esperados para una misma entrada. (a) Con las restricciones especificadas en [5]; (b) Con las muestras de [5] cuyas restricciones desconocemos.

El siguiente paso era crear nuevas figuras, Para ello hubo que hacer las figuras pixeladas, teniendo que emplear operaciones de Ansys para concatenar distintas figuras geométricas (distintos rectángulos para, por ejemplo, hacer un triángulo con bordes de sierra) o restas para hacer agujeros (que a veces podía ser más rápido).

Ansys nos permite juntar distintas figuras como si fueran una única pieza resultante. Así, a partir de la creación de rectángulos de diferente tamaños y de la suma y resta de estos, creamos a mano, una por una, 5 figuras nuevas diferentes.

En el artículo de referencia partían de 28 geometrías distintas. Nosotros, debido al tiempo limitado, hemos hecho 6 en total basándonos en las del artículo de referencia. Éstas son: el rectángulo, una con forma triangular, otra semicircular y las 3 anteriores con un agujero. En la Ilustración 16 se muestran algunas de estas figuras.

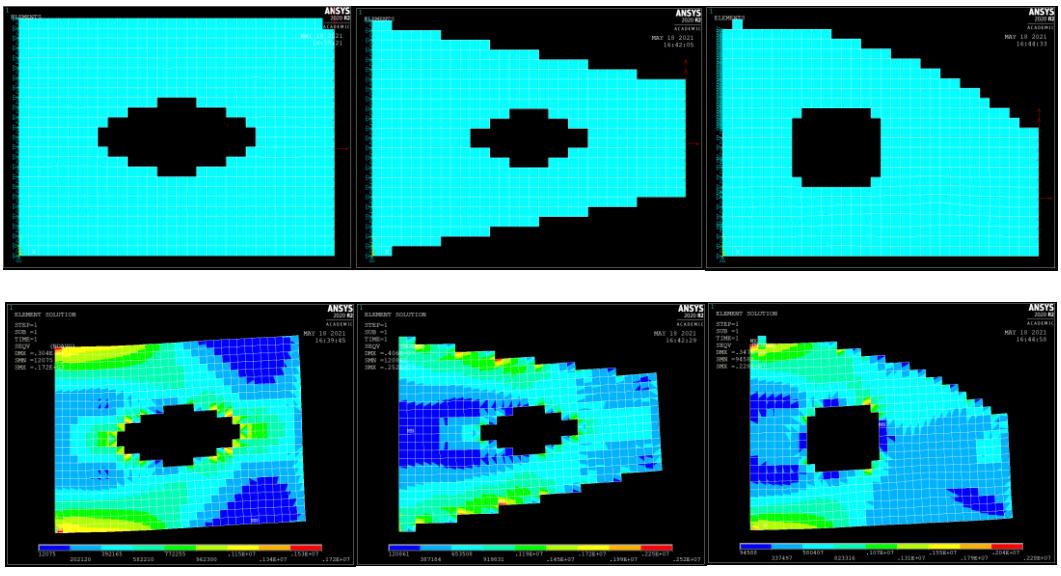

Ilustración 16: Figuras con agujeros desarrolladas con sus respectivas tensiones de Von Misses.

Una vez creadas las nuevas figuras, se modificó el programa en Matlab para que creara las 6 formas de las figuras y que, para cada figura, creara todas las muestras deseadas. De este modo se consiguieron 1681 muestras por figura, que dan un total de 10084 muestras de entrenamiento. Un esquema general del flujo de ejecución puede verse en la Ilustración 17. La creación de estas muestras ha llevado bastante tiempo de cómputo ya que sobrecargaba el ordenador del alumno y la generación de las muestras de 1 sola figura tardaba entre 1 hora y media y 2 horas, haciendo que se tuviera que ir dividiendo el proceso en lotes de figuras. Esta gran cantidad de tiempo se debe, principalmente, a la parte en la que Matlab está esperando el resultado por parte de Ansys.

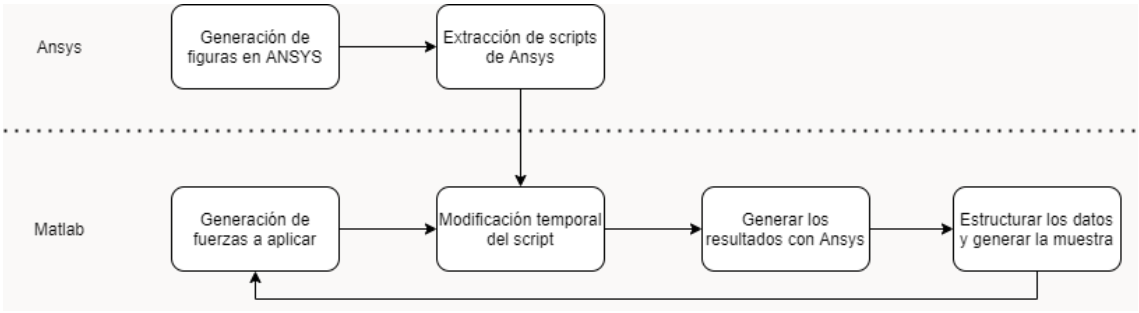

Ilustración 17: Flujo de la generación de muestras de entrenamiento.

También se ha tenido que comprobar que las muestras se crearan correctamente puesto que durante la realización del proyecto nos dimos cuenta de que a veces generaba siempre la misma figura con las mismas fuerzas. Este fallo ocasional se solucionó borrando los archivos temporales de cada muestra una vez se habían extraído los resultados deseados. Por algún motivo, a veces dejaban de sobrescribirse algunos archivos intermedios.

5.3. Experimentos y definición de hiperparámetros

Llegó el momento de utilizar nuestras muestras de entrenamiento para entrenar la red. Para ello subimos a Drive la carpeta con las muestras que conseguimos con Matlab y cargamos las muestras una por una en memoria para almacenarlas en un vector de 4 dimensiones (número de muestra, ancho, alto, canal). Una vez cargadas todas las muestras en el vector, guardamos en disco éste para no repetir este proceso posteriormente. Para ello transformamos el vector a un vector de tipo Numpy y usamos el método save. Cargar las muestras de esta forma requiere una gran cantidad de tiempo mientras que cargar la estructura tipo Numpy se hace en cuestión de segundos.

Una vez que tenemos el archivo de tipo Numpy estamos en el mismo lugar que al inicio del punto 5.1. pero con nuestras muestras. Con ello, pasamos a entrenar nuestra red con nuestras muestras de entrenamiento.

Antes de haber conseguido las 6 geometrías distintas, se probó a entrenar la red solo con la primera figura que se tenía (el rectángulo sin agujero). Se entrenó la red sólo con 2000 iteraciones y se vio que el resultado era muy malo. Tras este primer fracaso, se probó a hacer esas 2000 iteraciones en la red ya entrenada que se consiguió en el punto 5.1. pero sustituyendo ahora las muestras por las nuestras del rectángulo. Es lógico que al ser un problema parecido se podría llevar a cabo transferencia del aprendizaje y resultaría más eficaz y rápido entrenar sobre esta red que sobre una con los pesos inicializados de forma arbitraria. La diferencia en los resultados puede observarse en la Ilustración 18.

(a) (b)

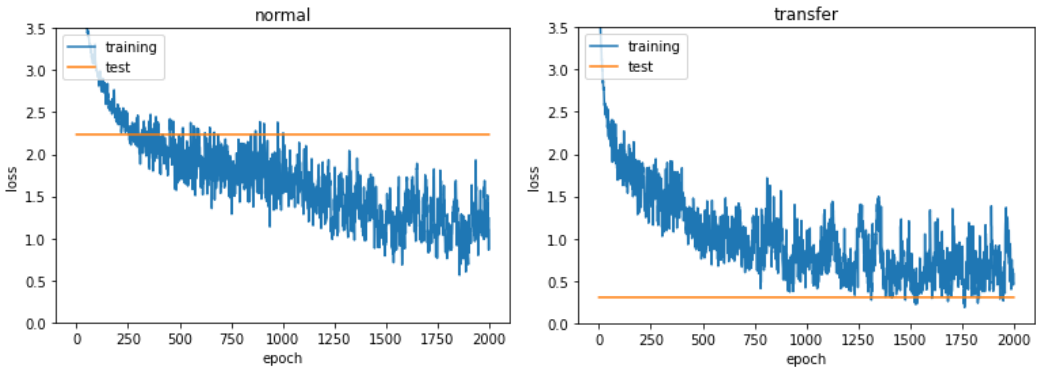

Ilustración 18: Entrenamiento y test con una sola figura. (a) Desde 0; (b) partiendo de la red entrenada en el punto 5.1.

Con esta misma figura, probamos a cambiar el problema, es decir, cambiar las posiciones de las fuerzas a aplicar. De este modo, se cambió la fuerza en el eje Y del lateral derecho a la cara superior y a la cara inferior, modificando únicamente en la entrada de la red el resultado que debe dar estas muestras y la posición de los unos en el canal de la fuerza Y (eliminando la fuerza anterior y añadiendo a la primera fila la magnitud fuerza y a la última fila la magnitud de la fuerza respectivamente). Teniendo ahora 3 tipos de muestras para 1 sola figura. Con esto, tratamos de entrenar nuestras más de 5000 muestras tal y como hicimos en el párrafo anterior: desde 0 y sobre la red ya entrenada en 5.1. El resultado fue que volvíamos a ver que la tendencia se repetía. Como puede observarse en la Ilustración 19, volvía a funcionar bastante mejor de nuevo la que se basa en la red ya entrenada. Ahora bien, a la hora de utilizar la red en muestras cuyo tipo de fuerza no era el original, el resultado era una matriz de ceros. Esto se debía a que cuando los resultados que se pretende conseguir son muy cercanos a 0 (en este caso lo son porque estas nuevas fuerzas tensionan menos la figura que la que aplicábamos originalmente) la red se dedica a colocar ceros en lugar de tratar de calcular los valores en cada punto. Así, el aplicar estas nuevas fuerzas se canceló puesto que implicaba aplicar fuerzas mucho más altas e incrementos mucho más altos para unos tipos de muestras, las que aplican este nuevo tipo de fuerzas, y otras más bajas para las muestras originales.

(a) (b)

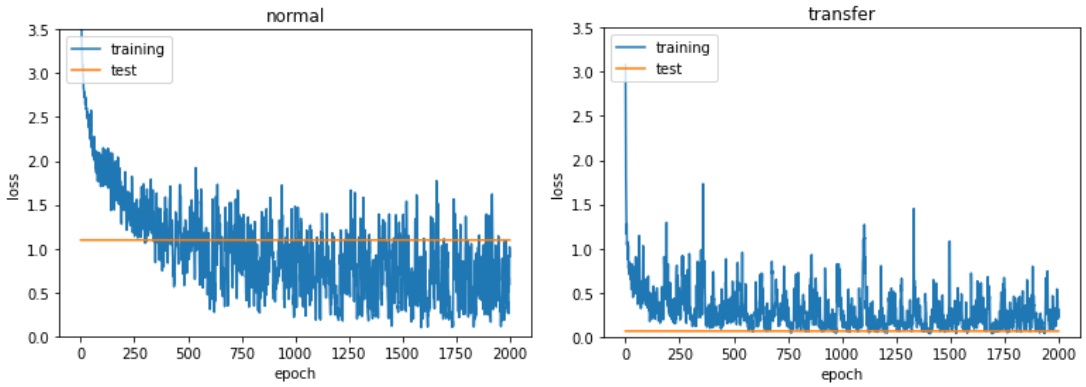

Ilustración 19: Error durante el entrenamiento y en test al aplicar 3 tipos diferentes de fuerzas a una pieza. (a) Entrenando la red desde 0; (b) partiendo de la red entrenada en el punto 5.1.

Podemos concluir también que con un bajo número de muestras e iteraciones interesa entrenar la red sobre la conseguida en el punto 5.1. en lugar que hacerlo desde 0. Esto en teoría no debería ser así conforme incrementemos el número de iteraciones y el número de muestras de entrenamiento.

Otro experimento que se hizo antes de empezar a entrenar una red fue comprobar como afectaba el tamaño del lote al funcionamiento de la red. Probamos así diferentes tamaños de lote con todas las muestras: 16, 32, 64, 128, 256 y 512.

Debido a la RAM limitada que teníamos (12GB) y a que al entrenar la red en Colab la RAM poco a poco se va saturando, llega un momento que cuando la ejecución llega al máximo disponible se para. Por ello, nos hemos visto obligados a hacer entrenamientos de 800 iteraciones, teniendo en cuenta que el cuello de botella estaba en el tamaño de lote de 64 que no llegaba a las 900 iteraciones sin antes abortar la ejecución.

El resultado de estos entrenamientos puede verse en la Ilustración 20, donde podemos observar que las oscilaciones del error se hacen más pronunciadas a medida que aumentamos el tamaño del lote. A su vez, el error en test va disminuyendo, hasta que llegamos a un tamaño de 128 donde conseguimos el error más bajo. Hay que mencionar que disminuir el tamaño del lote aumenta el tiempo de entrenamiento puesto que implica un mayor número de actualizaciones de los pesos de la red. De acuerdo con estos resultados usaremos un tamaño de lote de 128 para entrenar la red.

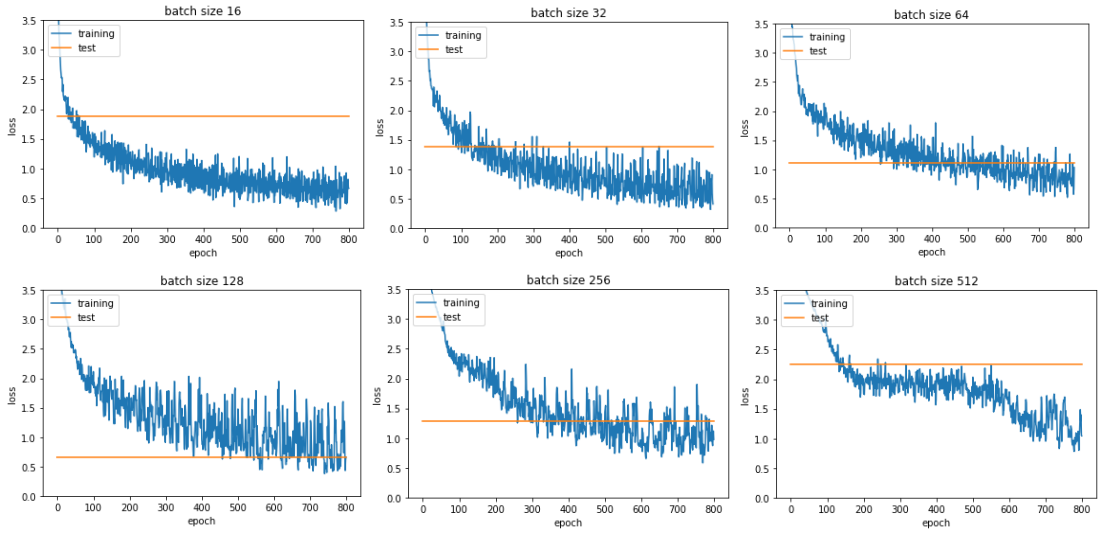

Ilustración 20: Errores de entrenamiento y test con diferentes tamaños de lote.

Por último, ya que no nos convence del todo ninguno de los experimentos anteriores comparado con el realizado en 5.1. con las 120000 muestras, dado que podría ser que la principal limitación fuera el número de muestras, hemos hecho pequeños experimentos de 1000 iteraciones tratando de transferir el conocimiento de la red 5.1. pero esta vez congelando algunas de sus capas en vez de entrenar sobre ella sin congelar ninguna capa. Se han ido congelando por orden las capas de forma acumulativa, es decir, congelando primero la primera, después las dos primeras, etc, así hasta que solo quedaba la última capa de la red sin congelar. Algunos de estos experimentos pueden verse en la Ilustración 21. A partir de las 9 capas congeladas el error es tan grande que deja de aparecer en las gráficas.

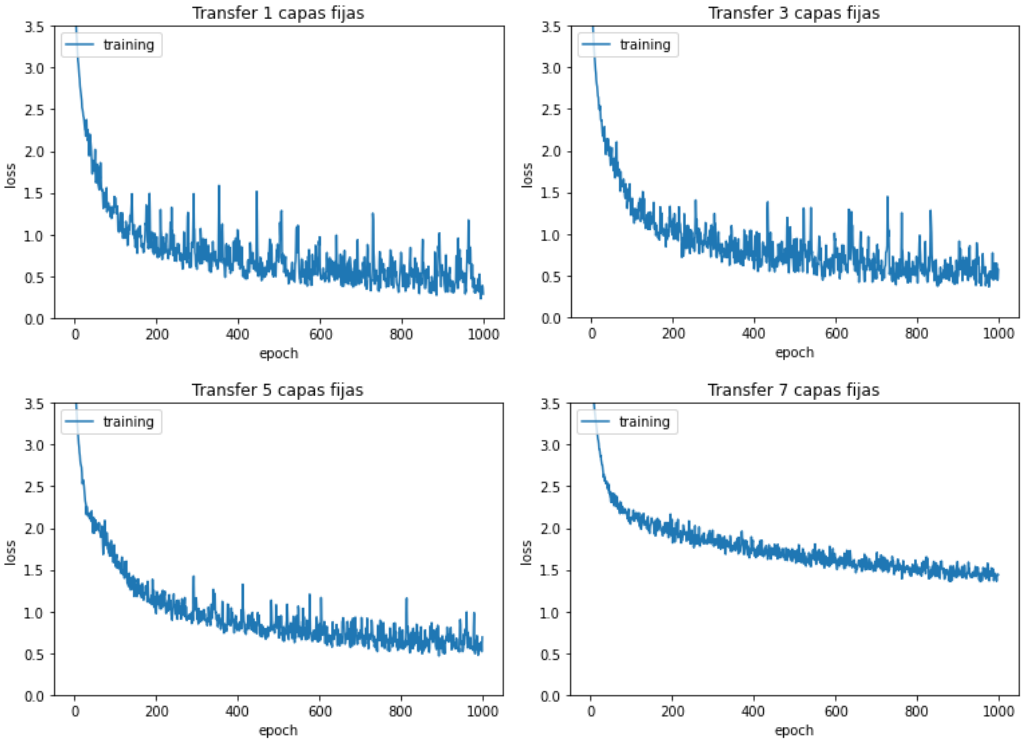

Ilustración 21: Errores al entrenar sobre la red del punto 5.1. al congelar las primeras capas.

En un principio debido a un fallo en el procedimiento, las capas SE-Res no se congelaban correctamente. Es decir, de estas capas solo se congelaba la normalización del lote que estas hacen, el resto seguía actualizando los pesos. Esta extraña forma de congelar, donde las 3 primeras y 3 últimas capas se congelaban correctamente, pero las capas intermedias solo congelaban la normalización del lote, dio lugar a algunos resultados sorprendentes al tratar de congelar todas las capas y todas las capas menos la última como puede verse en la Ilustración 22.

(a) (b) (c)

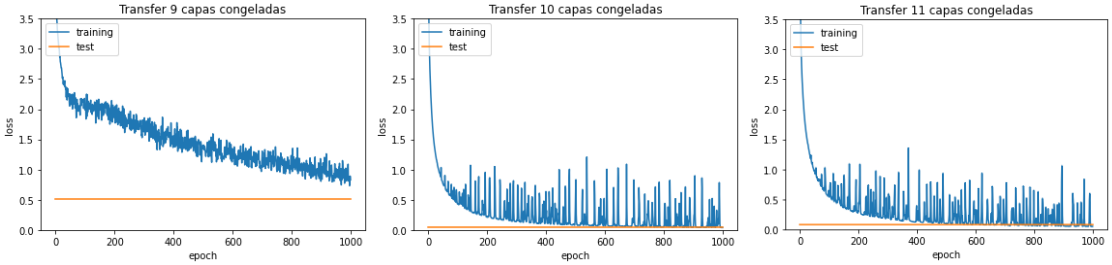

Ilustración 22: Error durante el entrenamiento y test congelando capas de forma errónea sobre la red con las muestras de [5]. (a) Congelando todas salvo las últimas 2 capas; (b) congelando todas salvo la última capa; (c) congelando todas las capas.

Vista la buena y rápida convergencia, esta particular forma de congelar se tendrá en cuenta a la hora de hacer futuros experimentos.

6. Resultados

Esta vez, debido a la RAM limitada que tenemos en Colab (en este punto disponemos solo de 12 GB de RAM en lugar de los 25 con los que contábamos en el punto 5.1.) nos hemos visto obligados a entrenar la red de 1000 iteraciones en 1000 iteraciones, llegando a unas 12000 iteraciones. Hemos elegido este valor puesto que ahora partimos de aproximadamente 10000 muestras en lugar de 120000 muestras, así obtenemos una red que haya actualizado sus pesos aproximadamente el mismo número de veces que en el experimento del punto 5.1.

Hay que destacar que Colab, al estar pensado para ser un entorno de ejecución interactivo, te restringe el uso de la GPU tras un tiempo determinado de uso. Por ello, hemos tenido que alternar las ejecuciones entre 2 cuentas. De este modo, cuando nos limitaba una cambiábamos a la otra y así sucesivamente. Otra alternativa, también utilizada alguna vez, ha sido simplemente esperar a que expirara la restricción que suele durar 24 horas.

Hemos hecho el experimento sin aplicar normalización debido a los resultados que obtuvimos en los experimentos previos explicados en el punto 5.1 y hemos utilizado 2 tamaños de lote, el original (256) y el que mejor funcionaba en los experimentos (128). Puede verse el resultado del entrenamiento con nuestras muestras en la Ilustración 23.

(a) (b)

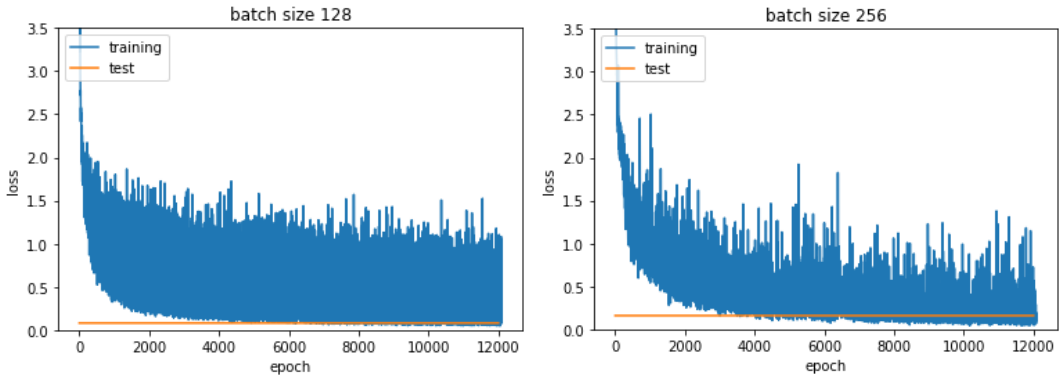

Ilustración 23: Error en entrenamiento y test para diferentes tamaños de lote.

En un principio los resultados finales eran bastante arbitrarios puesto que, como puede verse en la Ilustración 24, oscila mucho el valor del error. Para conseguir un buen error sin depender de la arbitrariedad de que la última iteración sea un pico o no se han hecho unas pocas iteraciones extra hasta que el entrenamiento para en un buen punto. Esto puede observarse en la Ilustración 25 donde sólo aplicando 80 iteraciones extra se ha conseguido disminuir el error en test desde 33.4738 hasta 0.2020 con el tamaño de lote de 128 y desde 2.3794 a 0.441 para el tamaño de

lote de 256. Todo esto únicamente porque se ha pasado de una iteración que terminaba en un pico de error a una que terminaba en un valle.

(a) (b)

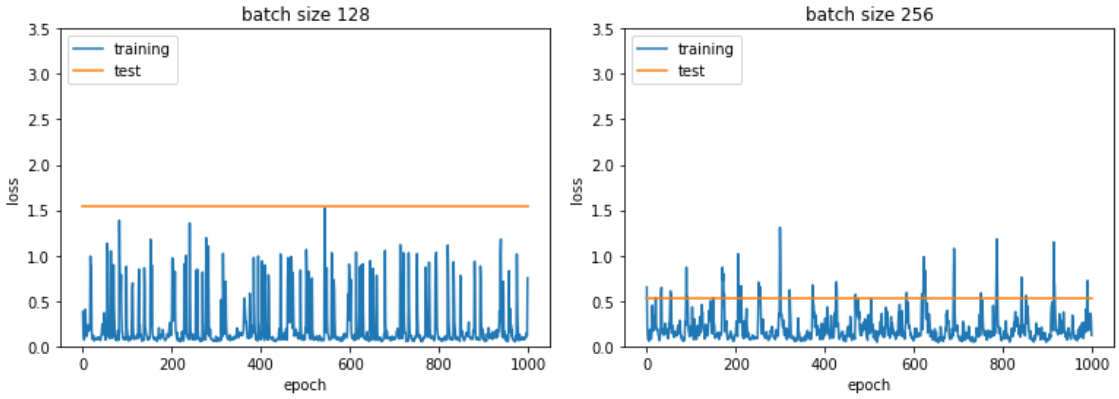

Ilustración 24: Logaritmo de los errores de test y entrenamiento en las últimas 1000 iteraciones del entrenamiento. (a) 4.6444 de error en entrenamiento y 33.4738 en test para el tamaño de lote de 128; (b) 0.3199 de error en entrenamiento y 2.3794 en test para el tamaño de lote de 256.

(a) (b)

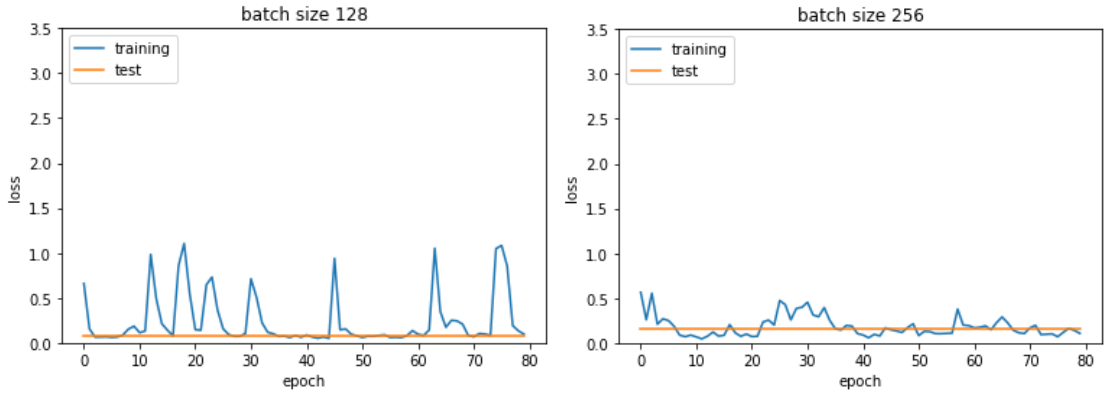

Ilustración 25: Logaritmo del error de test y entrenamiento en las 80 iteraciones extra para ambos tamaños de lote. (a) 0.2484 de error en entrenamiento y 0.2020 en test para el tamaño de lote de 128; (b) 0.2767 de error en entrenamiento y 0.44171 en test para el tamaño de lote de 256.

Puede observarse en la Ilustración 23, Ilustración 24 e Ilustración 25 que la oscilación de los valores de error es más suave para un tamaño de lote de 256 pero que el error en test es de más del doble y que para un tamaño de lote de 128 se llega mucho más rápido al error mínimo. Decir también que esta diferencia en el error en test no es significativa y no tiene por qué deberse únicamente a este hiperparámetro, puede deberse a que no se haya parado la ejecución en el mínimo más cercano para el tamaño de lote de 256. Por esto y porque converge mejor para un tamaño de lote de 256, se seguirá empleando este tamaño de lote de aquí en adelante.

Otra de las cosas que hemos probado es ver que ocurre cuando prescindimos de todas las normalizaciones de lote que hacemos a lo largo de la red. Este experimento se puede ver en la Ilustración 26, donde puede apreciarse como, tanto para lo bueno como para lo malo, se consiguen valores mucho más extremos, dando lugar a un error en test después de 6000 iteraciones de 0.007, dos órdenes de magnitud mejor que cualquier otro error que se ha obtenido en test, pero también dando lugar a una oscilación mucho más exagerada de los valores de error, donde prácticamente no puede apreciarse la convergencia de este.

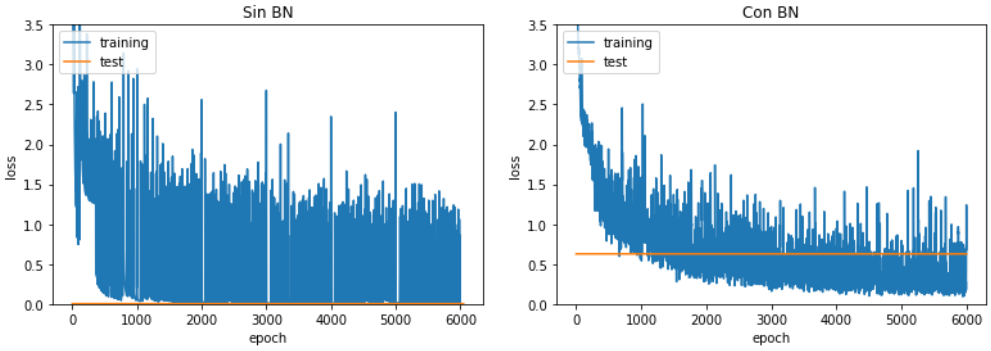

Ilustración 26: Diferencia en el entramiento aplicando normalización del lote y sin aplicarlo.

A continuación, puede observarse en la Ilustración 27 la diferencia entre la red entrenada con nuestras muestras de entrenamiento desde 0 (a), de la misma red al entrenarse con las muestras de [5] (c) y de la red entrenada sobre la red anterior con nuestras muestras congelando capas (b) (destacar que solo se han hecho 3000 iteraciones debido a lo rápido que converge). Como puede apreciarse los valores de la función de coste de la red entrenada con nuestras muestras oscilan mucho más que los de la red entrenada con las muestras del artículo. Creemos que esto se debe a la cantidad de muestras de entrenamiento y que podríamos llegar a conseguir que convergiera de forma más suave amentando estas. Esto se soluciona bastante al utilizar la red entrenada sobre las muestras del artículo congelando las capas de la forma que vimos en el punto 5.3

(a) (b) (c)

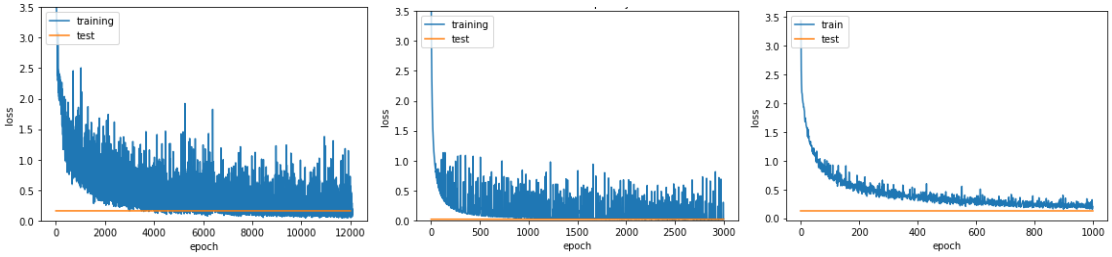

Ilustración 27: Error cuadrático medio en cada iteración durante el entrenamiento y después de este con el conjunto de test. (a) Con nuestras muestras de entrenamiento desde 0; (b) Con nuestras muestras de entrenamiento, partiendo de la red entrenada en el punto 5.1. congelando todas las capas salvo la última de la forma que vimos en el punto 5.3.; (c) con las muestras de [5] desde 0.

Otro motivo que aumenta la oscilación es la forma en la que lo estamos entrenando ahora que tenemos limitaciones de RAM. Parece ser que Keras, al entrenar un modelo, va disminuyendo el ratio de aprendizaje a medida que van haciéndose iteraciones. Es decir, que el error converge más lentamente para no ver oscilaciones muy grandes. Esto hace que al entrenar la red de 1000 en 1000 iteraciones aparezcan picos extras al principio de estos intervalos. Esto pude observarse claramente en la Ilustración 27 (a).

El error cuadrático medio de la red entrenada con nuestras muestras, al probarla con el conjunto de test, es de 0.441 mientras que el de la red entrenada con las muestras del artículo es de 0.363 y el de la red con las capas congeladas es de 0.041.

7. Conclusiones

Se han probado la red con tamaño de lote de 256 entrenada desde 0 y la red que partía de la del punto 5.1. con las capas congeladas. En la Ilustración 28 puede verse como el resultado de ambas redes para una muestra es idéntico al resultado esperado. Podemos ver que los resultados son similares, por lo que la diferencia en el error no es significativa.

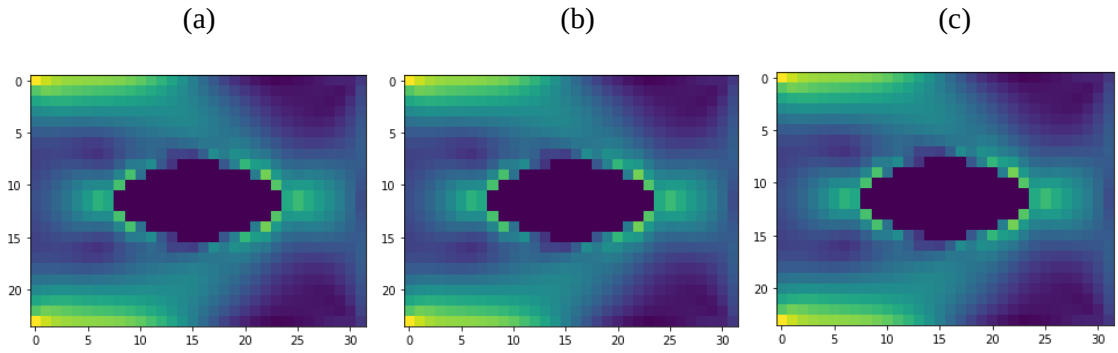

Ilustración 28: Resultado de una muestra de diferentes redes. (a) Tamaño de lote de 256; (b) tamaño de lote de 128; (c) resultado esperado

Gracias al experimento con diferentes tipos de fuerzas visto en el punto 5.3. nos dimos cuenta de que uno de los problemas de nuestra red es que cuando los resultados que se pretenden conseguir son muy cercanos a 0 la red se dedica a colocar ceros en lugar de tratar de calcular los valores en cada punto, como puede observarse en la Ilustración 29. Creemos que esto se debe a la función de coste penaliza muy poco este tipo de soluciones.

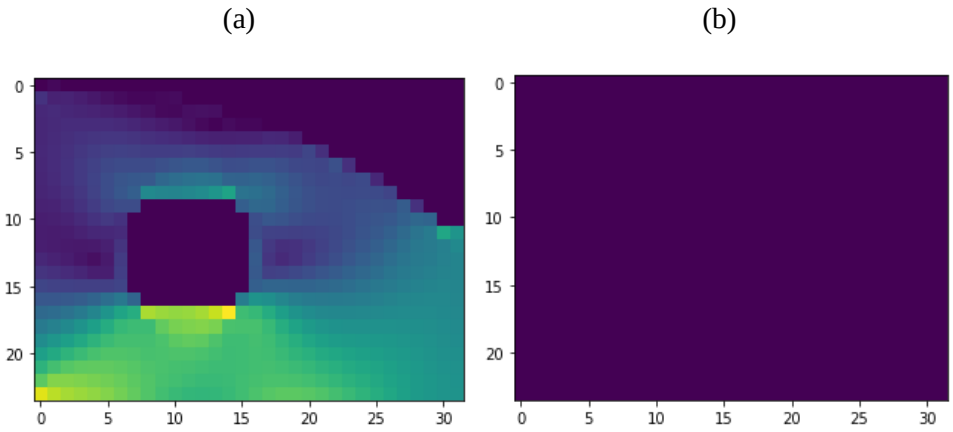

Ilustración 29: (a) Resultado esperado, donde los valores de la tensión de Von Misses oscilan entre 0.002 y 0.1; (b) Resultado de nuestra red.

Tanto al eliminar las normalizaciones del lote como al congelar estas dentro de las capas intermedias de la red se han conseguido resultados un orden de magnitud menor, por lo que, en este caso, probar diferentes configuraciones que involucren a esta capa parece la opción acertada.

Hemos utilizado la librería ipython-autotime para medir los tiempos de ejecución en Python. Tardamos 5.64 segundos en obtener la tensión de Von Misses para las 10000 muestras utilizando la CPU de Colab, si esto quisiera hacerse con Matlab llevaría en torno a 7 horas y media. Decir que con Matlab no solo calculamos la tensión de Von Misses y que tampoco creemos que ésta sea la forma más eficaz de lanzar múltiples ejecuciones en Ansys pero, aun así, la diferencia es de varios órdenes de magnitud.

Por todo ello, podemos concluir que nuestra red es capaz de calcular la tensión de Von Misses de estas figuras sin prácticamente perder precisión a una velocidad mucho mayor.

8. Trabajos futuros

Como trabajo futuro, se podría mejorar el programa ejecutado por Matlab haciéndolo más eficiente. Se podría sustituir la función que modifica el texto de los archivos para que en lugar de leer el archivo completo solo leyera las 3 primeras líneas que son las que se necesita modificar. También se podría automatizar la generación de la forma de las figuras tal y como se explicó en el apartado 5.2. para así no necesitar modificar el archivo de Matlab para lanzar nuevas figuras, solo se necesitaría añadir los nuevos ficheros .log a la carpeta correspondiente.

Otra de las claras mejoras sería buscar una función de coste que penalice más los resultados cercanos a 0 para resolver el problema de que para fuerzas muy pequeñas no funcione bien el modelo desarrollado. También se podrían añadir un mayor número de figuras para que la red aprenda a generalizar mejor y así funcione mejor con muestras incluso no conocidas.

Probar nuevas configuraciones que involucren, principalmente, a las normalizaciones del lote ya que modificar estas es lo que más a mejorado tanto el resultado de la red como la velocidad de convergencia del error.

A partir de lo mostrado en el trabajo se puede cambiar fácilmente el objetivo a calcular del modelo de red convolucional, diferentes tensiones, deformación de la figura, ... Solo habría que cambiar el resultado deseado por parte de Ansys y entrenar la red con los nuevos datos.

Por último, sería también interesante, ya desde el punto de vista de investigación, ampliar estre trabajo para que el modelo fuera capaz de simular el comportamiento de figuras en el espacio 3D.

9. Referencias bibliográficas

[1] Alom MZ, Taha TM, Yakopcic C, Westberg S, Sidike P, Nasrin MS, Hasan M, Van Essen BC, Awwal AAS, Asari VK, A State-of-the-Art Survey on Deep Learning Theory and Architectures, *Electronics*, 8 (292) (2019)

[2] H. Noh, S. Hong, B. Han, Learning deconvolution network for semantic segmentation (2015) 1520–1528

[3] V. Badrinarayanan, A. Kendall, R. Cipolla, Segnet: A deep convolutional encoder-decoder architecture for image segmentation, IEEE transactions on pattern analysis and machine intelligence 39 (12) (2017) 2481–2495.

[4] R. Valle, J. M. Buenaposada, and L. B., Cascade of Encoder-Decoder CNNs with Learned Coordinates Regressor for Robust Facial Landmarks Detection, Pattern Recognition Letters 136 (2020) 326-332.

[5] Z. Nie, H. Jiang, L. B. Kara, Stress field prediction in cantilevered structures using convolutional neural networks (2019)

[6] K. He, X. Zhang, S. Ren, J. Sun. Deep residual learning for image recognition.CoRR,abs/1512.03385 (2015)

[7] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks.CoRR, abs/1709.01507 (2019)

[8] J. Masci, U. Meier, D. Cire ̧san, and J. Schmidhuber. Stacked convolutional auto-encoders forhierarchical feature extraction. InInternational Conference on Artificial Neural Networks, 52–59. Springer (2011)

[9] F. Sultana, A. Sufian, P. Dutta. Evolution of Image Segmentation using Deep Convolutional Neural Network: A Survey (2020)

[10] A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, (2012)

[11] Z. Cai, N. Vasconcelos, Cascade R-CNN: Delving into High Quality Object Detection, IEEE/CVF Conference on Computer Vision and Pattern Recognition 6154-6162 (2018)

[12] D. H. Hubel, T. N. Wiesel, Receptive fields and functional architecture of monkey striate cortex, Journal of Physiology (London) 195 (1968) 215–243

[13] K. Fukushima, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, Biological Cybernetics 36 (4) (1980) 193–202. doi:10. 1007/BF00344251.

[14] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, Back-propagation applied to handwritten zip code recognition, Neural Comput. 1 (4) (1989) 541–551. doi:10.1162/neco.1989.1.4.541.

[15] S. Ioffer, C. Szegedy, Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift, (2015)

[16] D. P. Kingma, J. L. Ba, Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, (2015) 1–13.