Using Artificial neural networks to predict patterns in C. elegans’ movement

Procesos seguidos en las prácticas de Enrique Galceran

Al principio hemos creado una pequeña red neuronal excepcionalmente básica pero que sepamos que funciona, con el objetivo de que después podamos meterle la base de datos del movimiento. La ventaja de empezar con una red más pequeña es que en el caso de que no funcione la primera vez podamos saber que efectivamente viene causado porque la entrada de datos no es del todo buena y no debido a nuestra red.

Hitos en la primera red (SimpleNN):

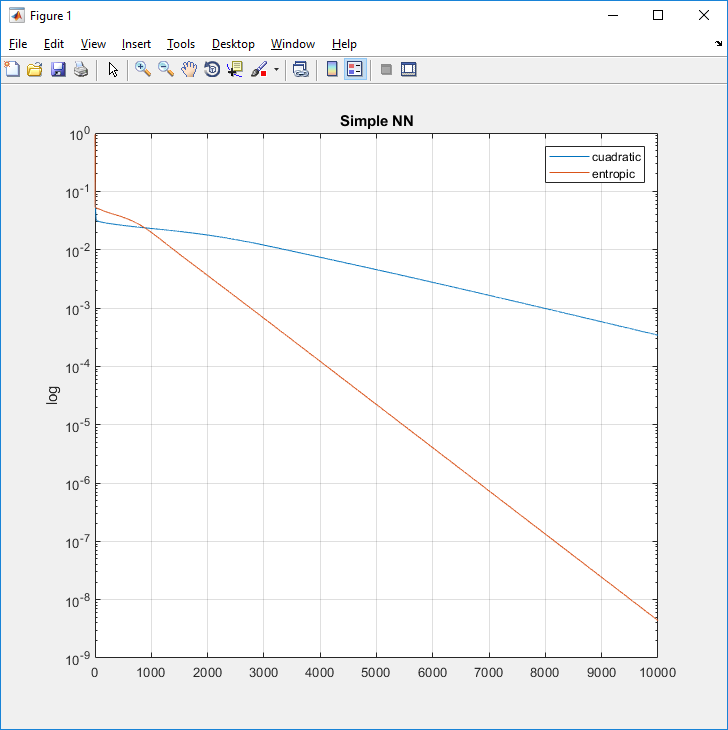
1. **Primeras predicciones**. Ha sido capaz de empezar a predecir correctamente la base de datos inicial. Dados dos entradas (horas estudiadas y horas dormidas) puede predecir el valor de una función final (nota del examen). Esta primera versión utilizaba una versión simplificada de mínimos cuadrados para obtener la función coste (Y-resultados).
2. **Introducción del bias**. En toda Red Neuronal grande es recomendable introducirle un valor llamado ‘Bias’. Esta función cambia el baseline con el que se va a activar nuestra neurona. A niveles de computación, tiene el mismo efecto que desplazar nuestra función de activación hacia la izquierda o a la derecha.
3. **Learning Rate**. Llegado a un punto más avanzado, la red dejaba de converger. Esto se debía a que el ‘gradient descent’ que estaba siguiendo nuestra red eran demasiado grandes. Introducirle la variable Learning Rate (LR de ahora en adelante) modifica el tamaño de los pasos dados en cada iteración. Como LR<<1 para i>>1, reducimos el tamaño de estos pasos a medida que nos vamos acercando al resultado final y la función coste se reduce en una gran cantidad.
4. **Introducción de los datos de los gusanos.** Una vez teníamos una NN que funcionase, le introdujimos los datos experimentales del primer gusano. Como era de esperar, convergía mucho más despacio que nuestras redes anteriores, hasta llegado el punto en el que parecía que literalmente no iba a converger en ningún momento.
5. **Vanishing gradient problem.** El problema de usar funciones Sigmoides es que, para valores muy grandes, su derivada se hace prácticamente nula. Esto ha sido uno de los mayores problemas desde el primer minuto. Tenemos una red muy grande con muchas capas, lo que genera valores muy grandes con los que trabajar-> derivadas muy pequeñas. Aumentar el LR al principio ayuda bastante con esta parte, pero si lo aumentamos demasiado genera desestabilidades muy grandes que causan más problemas que ayudas.  
   Una de las ayudas propuestas es cambiar los valores iniciales de las sinapsis. Erróneamente, habíamos dado como valores iniciales para dichas sinapsis (0,1), luego siempre aumentaba y tardaba mucho en converger. Cambiar los valores iniciales a (-1,1) ayudó mucho, pero no terminaba de desaparecer este problema. La solución final para recortar mucho la fase inicial donde la red está andando como gallina sin cabeza consistió en sustituir la generación de números aleatorios lineales a una distribución normal μ=0, σ=1. La solución oficial propone una mejora a esta inicialización, consistente en utilizar una normal μ=0, . La ventaja de usar esta normal se ve claramente cuando tenemos una red con muchas neuronas por capas. Por mucho que los puentes tengan de media un valor de 0 y se cancelen al principio, hay una probabilidad no nula de que los valores iniciales superen el ±4, y sigmoide(4)= 0.982, que para nuestro primer valor inicial es muy grande. Aunque a primera vista tenía buena pinta usar esta nueva distribución normal, en la práctica los resultados son demasiado cercanos a 0 y le cuesta mucho ir a cualquier lado.
6. **Backups.** Como no estaba funcionando del todo bien el ajuste y le estaba costando, empecé a dejar el ordenador trabajando por la noche para aprovechar ese tiempo de cálculo. Creé dos variables ‘cada\_cuanto\_guardar’ y ‘cada\_cuanto\_limpiar’. La función de esas variables es definir los rangos en los que se crea un archivo ‘.mat’ con las diferentes variables para que en el caso de que se cuelgue el programa no se pierda todo ese trabajo hecho y se pueda recuperar esos valores. Después me di cuenta de que se podría aprovechar estos archivos de guardado para que se pueda crear una cronología de lo que ha ido pasando. Después de crerar los scripts de Comprobar y Análisis de tarta empecé a guardar los datos resultantes de las funciones comprobar dentro de estos archivos. La segunda variable (limpiar) tiene como objetivo liberar espacio del disco duro a medida que vaya avanzando. Nos da igual a la hora de analizar lo que ha pasado si tenemos 3000 o 300 o 100 valores a la hora de realizar el gráfico. Esta función limpia cada ‘limpiar’ veces que se guarda todos los datos intermedios creados hasta la última vez que se borró.

Así pues, si tenemos los valores 1000 y 10, cada vez que i = k\*1000 donde k ∈, borrará todos los que haya entre i= (k-1)\*1000 y i=k\*1000 sin incluir los límites.

1. **Comprobar.** Teniendo en cuenta la naturaleza discreta de nuestros datos (tanto los de entrada como los de salida), resulta bastante difícil apreciar si los resultados que estamos obteniendo en nuestra red son buenos o malos. Para corregir eso, lo que hemos creado es un pequeño script que dada una red neuronal (sus sinapsis) genera una entrada a partir de la base de datos y compara cada uno de los 90 resultados con los esperados. En una red perfecta, tendría 89 ceros de salida y 1 sólo uno. Es por eso que cada una de las salidas podemos clasificarla en una de cuatro categorías: 0 que debe ser 0, 1 que debe ser 1, 0 que debería ser 1 y 1 que debería ser 0.
   1. La primera de las categorías las vamos a ignorar, pues en el caso ideal eso sería un número muy grande que sería poco práctico que representar con los demás; pudiendo discernir ese valor a partir del resultado de los otros 3.
   2. El segundo valor será nuestra categoría de resultados correctos. Esa es la que queremos que tenga el mayor valor posible (1 por cada línea de entradas, luego en el caso perfecto correc\_sum = len\_data y ambos errores van a tener un valor de 0)
   3. El tercer y cuarto valor forman parte de los resultados incorrectos. Para poder hacernos una mejor idea de donde se encuentra el error mientras que está aprendiendo la red las tenemos separadas en falso positivo y falso negativo. Más adelante (en el punto 12) hablaré más a fondo de la relevancia de poder distinguir cuánto se está equivocando en cada momento.
2. **Función entropía.** La base matemática no la termino de comprender del todo bien, pero cambiando la función de coste lineal o la cuadrática son mucho menos eficientes que esta nueva función de Coste:

Donde ‘A.’ representa realizar las operaciones sobre la matriz A elemento a elemento.

Hacer esto reduce drásticamente el coste de la red y podemos ver como inicialmente tarda un poco más, pero en seguida converge muchísimo más rápido:



1. **Análisis de la tarta.** Inicialemente surgido como un ‘pun’, debido a la utilización de un pie-chart para representar lo bien o mal que se está ajustando en cada momento a los valores esperados, la variable que guarda los valores de dicha gráfica se llamaba ‘tarta’. Como las funciones de backups guardaban también esa variable, podíamos ponernos a representar la variable tarta en función de la iteración y ver a la progresión en el tiempo de bien o mal que se ajustaba.
2. **Reestructuración de la base de datos (V9).** Después de bastantes versiones diferentes del código, me fijé en que, aunque estuviera funcionando, se perdía mucho tiempo en el tratamiento de las entradas que se podría realizar previamente de golpe (y más rápidamente/de forma más eficaz), ahorrando así un precioso tiempo en la fase de aprendizaje. Para ello voy a centrarme en las tres partes de esta sección:
   1. **Inconsistencia en los nombres.** La base de datos original tenía un nombre diferente para cada gusano que iba desde 2775 hasta 4036. Sin embargo, hay ~80 de esos gusanos que no están contabilizados en la base de datos. Eso ha creado algún que otro problema en el código al intentar leer un archivo inexistente. Por eso, usando el script ‘limpiar\_nombres.m’ podemos renombrar los nombres de estos archivos y ponerlos con un formato que nos convengan más (empezando por 0001 y terminando en 1188)
   2. **Preparación de las entradas.** Cada iteración cogía la secuencia de números y generaba dos vectores unidimensionales con 90 valores que representan las diferentes plantillas. Lo que hacemos nosotros es crear una nueva base de datos con todos los valores de las plantillas ya preparadas con la función peso.
   3. **Mezclar las plantillas que se van a utilizar.** Nuestra red neuronal utiliza el patrón vector->red->vector para cada iteración. En cada iteración la salida nos dice en que dirección queremos empujar la red (ligeramente) para que se ajuste mejor a los resultados. Después de unas cuantas iteraciones, aquellas posiciones que son más sensibles a cambios habrán aumentado y disminuido, teniendo varias entradas de datos que van a trabajar en dirección contraria. Hay otras partes de la red en las que las después de varias iteraciones la mayor parte de ellas han empujado en la misma dirección. Para acelerar el proceso, vamos a utilizar el sistema ‘batch-processing’:

Bath-processing se aprovecha de que podemos utilizar el patrón matriz->red->matriz y así procesar varias líneas de entrada a la vez. Esto suaviza la forma en la que se modifican cada sinapsis de la red iteración tras iteración. No va en una línea tan recta hacia el mínimo de la función de costa (recordamos que es el objetivo de la red), pero va más rápido a nivel global.

* 1. **Reescribir el código para acelerar el proceso y ajustarse a la nueva forma de meter datos.** Bastante auto explicativo. Menos importante o trascendental, pero hay que hacerlo. La nueva forma de meter datos significa que vamos a tener que reescribir parte del código para que se ajuste a eso. La ventaja es que ahora podemos configurar de una forma muy sencilla y rápida la cantidad de gusanos (y cuáles de ellos) que queremos que se estén utilizando en cada momento para la simulación.
  2. **Comprobar2.** Ya que hemos creado una forma de analizar y crear redes para múltiples gusanos, ya puestos hemos creado otra forma para analizar la eficacia de esta nueva red. Esta mira si funciona bien con ambos gusanos, no solamente con uno.

1. ***Markov*.** Uno de los objetivos de la práctica es comparar el sistema de las redes neuronales a la hora de predecir la siguiente posición comparado con las predicciones que obtenemos con Markov. Para ello, tras generar las matrices de transición de Markov, creamos la función de probabilidad acumulada y creamos varias simulaciones para ver si predice correctamente la siguiente posición.
2. ***Comprobar3*.** Uno de los objetivos futuros es reestructurar la forma en la que evaluamos los resultados de nuestra red. Ahora mismo estamos redondeando. En una red perfecta debería dar resultados altos para solamente uno y todos los demás deben tener valores bajos. Sin embargo, es muy probable que estemos teniendo falsos positivos debido a que estamos redondeando (tomando como valores para éxitos >0.5). Esta es la razón por la que distinguimos entre falsos Positivos y falsos negativos. Si no lo hiciéramos, no distinguiríamos entre resultados que deberían ser 0 y resultados que deberían ser 1. Aunque ambos casos sean fallos en la predicción, para realizar la red neuronal es interesante poder saber cual es la causa del error: los FP se corrigen con el tiempo y aprendizaje y los FN después de muchas iteraciones son más problemáticos de que desaparezcan. Este script debería hacer lo mismo que Comprobar y Comprobar2 pero solamente tomando como salida el mayor de los valores de la salida. Me espero que los resultados de Comprobar2 deben ser radicalmente diferentes a Comprobar3.
3. ***Dibujar la propia red y sus sinapsis.*** Uno de los objetivos que tenía desde el primer día es dibujar la red y sus sinapsis. El problema está en que probablemente no sea la mejor idea, pues tenemos 500 neuronas con 34.000 sinapsis, y dibujar eso no tiene mucho sentido, pues no se apreciaría absolutamente nada.

Comentarios que he ido observando:

La forma de meter los datos ha sido la principal fuente de problemas. Por un lado está muy bien el tener que trabajar con sólo 90 posiciones para trabajar con todas las diferentes posiciones que puede tomar el gusano. Por otro lado, esas 90 posiciones son posiciones discretas y por regla general las redes neuronales no son muy fan de entradas discretas. Salidas discretas no hay ningún problema. Dicho eso, parte de la gracia de generar esta red neuronal está en poder compara los resultados con Markov. Aunque Markov sea claramente más rápido de procesar, es muchísimo menos eficiente. Después del rato que he estado escribiendo esto, la red ha sido capaz de ajustarse con un 40% de éxito\* a los datos de **dos** gusanos, mientras que usando Markov para solamente un gusano sólo obtenía un 19% de éxito. Las redes menos eficaces que hemos estado creando en los días anteriores (V4, V8b y V8c) eran capaces de sacar redes neuronales con una eficacia de 90%-95% consistentemente después de 50min de trabajo! Si no hay prisa en crear la red neuronal, estas son muchísimo más eficaces en ser capaces de predecir el movimiento comparado con Markov.

Sí que es verdad que tiene problemas generar estas redes neuronales. Hay muchos parámetros a tener en cuenta para calcular todas las redes. Si haces las redes suficientemente grandes como para que se ajusten bien, tardan muchísimo en avanzar y corregirse, pero si las hacemos demasiado pequeñas nos arriesgamos a que no sean capaces de ajustarse.

Una de las principales razones por la que son tan eficaces las redes neuronales grandes es porque utilizan capas de convoluciones (Convolutional NN). Estas utilizan como entrada unos cuantos valores contiguos de la entrada. La ventaja de hacer eso es que da una mejor información de los valores en forma de ‘clusters’ en vez de pixel a pixel. Al reducir la cantidad de entradas, pero teniendo una mayor cantidad de información en cada uno de ellos, se ajusta más rápido al valor objetivo. El problema con esta red es que no se puede realmente hacer eso. De momento me he estado centrando en otros aspectos de la red que se pueden mejorar de una forma más directa y rápida, para así asegurarme de que al menos estoy obteniendo algo. Podría intentar organizar las diferentes plantillas en grupos más pequeños y crear así las convoluciones. Técnicamente no serían convoluciones, porque todas las entradas (salvo los bordes si hablamos de procesado de imágenes) aparecen en muchas convoluciones. Si agrupase las diferentes plantillas en grupos más pequeños, esta pseudo-convolución generaría una capa reducida, pero probablemente no tenga tanta información como sería la otra.

Me he ido por las ramas. Resumen del párrafo anterior: Las convoluciones son interesantes de implementar. Normalmente se generan 3 capas de convolución (todas aleatorias al principio), pero sólo tiene sentido hacerlas si las entradas son continuas. Para las entradas discretas (como en nuestro caso) no tiene mucho sentido realizar convoluciones sobre ellas, pues tendríamos que hacer grupos con separaciones bien definidas y no tendría una separación “difuminada”.

Como cosa bastante aparte, se me ha ocurrido una posible aplicación de esta red: si necesitamos comparar entre varios tipos diferentes de gusanos, podemos crear una hiper-red muy densa y profunda que vale para un tipo de gusanos (A) y use una enorme cantidad de datos para aprender; y después creamos una segunda red para el otro tipo (B). Después podemos utilizar utilizar estas redes para ver lo bien o mal que se ajustan a un tipo o al otro. Sin embargo, para eso podemos simplemente meterle dos salidas (A) y (B). con ello podemos crear una red que clasifique. Una vez tenemos eso, podemos cambiarle los valores de entrada hasta que llegue a un punto en el que la red no sea capaz de diferenciarlos. Esta es una modificación de lo que hablamos en su momento.

La otra aplicación podría ser para aquellos casos en los que tenemos que predecir el movimiento que va a hacer un gusano de forma automática para ajustar la máquina rápidamente.

La tercera opción, pero que lamentablemente no tengo los datos suficientes, es grabar a un gusano haciendo dos cosas diferentes (por ejemplo, colocarlo en una placa de agar vacía y que busque comida y por otro lado uno en el que pueda seguir un gradiente de comida y se esté moviendo hacia él). Las redes neuronales que se ajustan a un solo gusano muy buenas, y cuando comparamos los resultados de una red con un gusano que no es el “dueño” de dicha red, se puede apreciar enseguida cómo la eficacia de predicción baja drásticamente. Es por eso que podemos (en teoría) crear dos redes para los dos casos diferentes y podemos comprobar a ver si podemos predecir de forma cruzada. Sería interesante ver si un gusano deja de ajustarse a la red que ha sido entrenada específicamente para él cuando está haciendo dos acciones diferentes. ***¿Cuál es el límite para el que deja de ajustarse?***

Cosas que debería mejorar pero no se del todo cómo/dónde meterle mano:

1. Quiero meterle como función de activación ReLU, pero tengo el problema de que siempre me explota y no se cómo corregirlo. En todas partes dice que es mucho mejor que las demás (Sigmoide, te miro a ti) para redes neuronales grandes porque a) es más rápido y b) no tiene el problema del vanishing gradient.
2. Quiero hacer capas de convolución. Dice que es mucho mejor hacerlas, pero realmente sólo tienen sentido para casos en los que las variables sean continuos. Arriba hablo en mayor detalle sobre ello.