Redes neuronales - Proyecto final - Programación avanzada

Nicolás Aylwin

1. Introducción

Este proyecto se centra en el desarrollo de redes neuronales artificiales capaces de replicar valores deseados a través de entrenamiento supervisado (entregar lista de datos "input" y un "output" deseado). Para esto se desarrolla una clase de capas neuronales lo que facilita el uso de operaciones matriciales para el desenvolvimiento de la matemática subyacente a los algoritmos necesarios.

Los programas incluyen una serie de detalles que facilitan su uso reiterado y personalizado, además de una serie de ejemplos (en especial visuales) de comportamiento de las redes creadas.

2. Teoría

2.1. La red

Las redes neuronales artificiales tratan de simular (a nivel de algoritmo y data structure) la complejidad de conexiones en nuestro cerebro, e idealmente, lograr de igual manera tareas que parecieran ser únicamente realizables por humanos. Está claro que la idea ha sido un éxito, siendo imposible no relacionarse (de forma indirecta al menos) con estas en el mundo moderno. De forma simple, nos podemos imaginar su estructura de la siguiente manera:

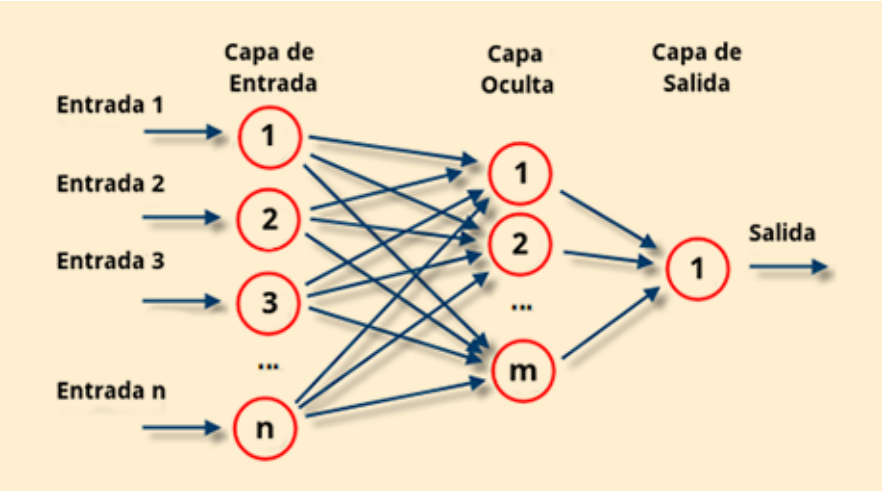

Figura 1: Esquema de red neuronal

En Fig.(1) podemos distinguir varias neuronas (los circulos) agrupadas en tres tipos de capas principales; la capa de entrada (que recibe los datos), la capa oculta, y la capa de salida. En general, se pueden tener una cantidad arbitraria de capas ocultas, así como cualquier cantidad de dimensiones para datos de entrada y datos de salida.

2.2. Cálculo: Proceso Forward

Para procesar la información, cada neurona realiza dos pequeñas operaciones. Primero una operación lineal, de manera que si la neurona N_i recibe un dato de dos componente x_1 y x_2, retorna una suma ponderada más una constante de la forma $\sum_{j=1}^{2} a_{ij}x_j + b_i$ con a_{ij} y b_i reales. La segunda operación se le llama "función de activación", que consta de agregar no-linealidad al output final de cada neurona, de manera que el output final y_i de la neurona N_i si es que la función de activación es $f(x)$ sería $y_i = f(\sum_{j=1}^{2} a_{ij}x_j + b_i)$.

Es claro ver que este proceso se puede ver como operaciones matriciales si consideramos operaciones por capas en vez de neuronas. Considerando un input de tres componentes y un primer encuentro con una capa de dos neuronas podríamos representar la operación de esta capa como:

$$f\left(\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right) = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

Entendiendo que f actúa sobre cada elemento del vector resultante.

Consideremos la operación equivalente:

$$f\left(\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} + \begin{pmatrix} b_1 & b_2 \end{pmatrix}\right) = \begin{pmatrix} y_1 & y_2 \end{pmatrix}$$

Puesto este formato de datos es más usual, por lo que conviene trabajar de esta manera. Por otro lado, consideremos si es que se entregan múltiples datos simultáneamente:

$$f\left(\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots \end{pmatrix}\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} + \begin{pmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \end{pmatrix}\right) = \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \\ \vdots & \vdots \end{pmatrix}$$

Nótese que a_{ij} y b_i son los mismos para cada dato (de dimensión arbitraria, tres en este ejemplo), pues son elementos propios de la neurona.

Si encadenamos que el último output sea el input de la siguiente capa tenemos una forma general de procesar todos los datos entregados de forma simultánea a través de la red, controlando las dimensiones de los outputs a través de la cantidad de neuronas en cada capa, pudiendo regular así facilmente las dimensiones del output final.

Pero ¿cómo podemos lograr que todas estas operaciones logren generar los resultados que esperamos de los datos de entrada?. Para eso es necesario ajustar los a_{ij} y b_i (que desde ahora llamaremos pesos y constantes) de forma inteligente.

2.3. Ajuste de red: Proceso Backward

Utilizaremos algoritmo de backpropagation y descenso del gradiente para modificar la red. Basta comparar los resultados obtenidos y con los resultados esperados, llamemos a estos Y (ambos en general matrices). Usando alguna función $E(y, Y)$ de error cualquiera podemos comparar y modificar los pesos y constantes de cada neurona. Consideremos primero la última capa N de una red neuronal, y estudiemos solo un dato de dimension arbitraria. Para esto definamos O_j como el output, P_j como la matriz de pesos, y K_j como el resultado de la suma ponderada más la constante (todas de una capa j). Es claro que $O_j = f(K_j) = f(O_{j-1}P_j + b_j)$ con f la función de activación.

Así si buscamos minimizar el error para la última capa, tomando sus derivadas y usando regla de la cadena:

$$\frac{\partial E}{\partial p_N} = \frac{\partial E}{\partial o_N}\frac{\partial o_N}{\partial k_N}\frac{\partial k_N}{\partial p_N}$$
$$\frac{\partial E}{\partial b_N} = \frac{\partial E}{\partial o_N}\frac{\partial o_N}{\partial k_N}\frac{\partial k_N}{\partial b_N}$$

Donde las minúsculas representa que son elementos de las matrices.

Se tiene que las dos primeras derivadas en ambas ecuaciones son la derivada de la función de error respecto al output final de la neurona (pues $O_N = \vec{y}$) y la derivada de la función de activación. Para las otras dos es claro que:

$$\frac{\partial k_N}{\partial p_N} = o_{N-1} \quad ; \quad \frac{\partial k_N}{\partial b_N} = 1$$

Además es usual definir las dos primeras derivadas en ambas ecuaciones como δ_j, y definirlo como el error asociado a la capa (o a la neurona dependiendo de como se realicen las operaciones). Así podemos definir nuevamente las derivadas como:

$$\frac{\partial E}{\partial p_N} = \delta_N o_{N-1}$$
$$\frac{\partial E}{\partial b_N} = \delta_N$$

Calculando ahora para la penúltima capa se puede conseguir una expresión general para "retropropagar" el error. Primero se tiene:

$$\frac{\partial E}{\partial p_{N-1}} = \frac{\partial E}{\partial o_N}\frac{\partial o_N}{\partial k_N}\frac{\partial k_N}{\partial o_{N-1}}\frac{\partial o_{N-1}}{\partial k_{N-1}}\frac{\partial k_{N-1}}{\partial p_{N-1}}$$
$$\frac{\partial E}{\partial b_{N-1}} = \frac{\partial E}{\partial o_N}\frac{\partial o_N}{\partial k_N}\frac{\partial k_N}{\partial o_{N-1}}\frac{\partial o_{N-1}}{\partial k_{N-1}}\frac{\partial k_{N-1}}{\partial b_{N-1}}$$

Las dos primeras derivadas corresponden al error de la capa siguiente (N),la tercera a la matriz de pesos (P_N), la cuarta es la derivada de la función de activación y las dos últimas son equivalentes a las últimas en el caso anterior (o_{N-2} y 1 para esta capa). Definimos ahora el error de esta capa (y de todas las que quedan) como las cuatro primeras derivadas.

De esta manera se generaliza el cálculo de los errores como:

1) Calcular error de la última capa:

$$\delta_N = \frac{\partial E}{\partial o_N}\frac{\partial o_N}{\partial k_N} \tag{1}$$

2) Retropropagarlo a la capa anterior (general para cualquier L menor a N):

$$\delta_L = P_{L+1}\delta_{L+1}\frac{\partial o_L}{\partial k_L} \tag{2}$$

3) Luego calcular así las derivadas de la capa:

$$\frac{\partial E}{\partial p_L} = \delta_L o_{L-1} \quad ; \quad \frac{\partial E}{\partial b_L} = \delta_L$$

Generalizando para la cantidad de datos, la estructura en que los tenemos y usando descenso del gradiente se tendrá, para la última capa:

$$\hat{P_N} = P_N - \lambda(Y^T * \delta_N)$$
$$\hat{b_N} = b_N - \lambda < \delta_N >$$

Donde las expresiones de la izquierda representan los nuevos valores para las matrices de pesos y constantes, λ número positivo que corresponde al ratio de aprendizaje (o paso), Y ya enunciado al principio de esta sección, δ_N es matriz con cada elemento siendo resultado de la Ec.(1), filas equivalente a la cantidad de datos resultantes, columna equivalente a la dimensión de estos resultados y $< \delta_N >$ es vector fila con cada elemento siendo el promedio de cada columna de δ_N.

Toda delta δ_L con $L < N$ será una matriz con cada elemento equivalente a Ec.(2), siendo δ_{L+1} la matriz δ calculada en la iteración anterior. Habiendo realizado esto, los nuevos pesos y constantes se calculan con la misma fórmula de los de la última capa, solo que sustituyendo el δ por el correspondiente a la capa, e Y por los outputs de la misma.

Realizando esto una cantidad suficiente de veces (dependiendo de la cantidad de datos, dificultad del problema y precisión buscada), se logra minimizar el error a través del descenso del gradiente, y por ende, los outputs obtenidos convergen a los resultados esperados.

3. Programa

3.1. Clase de capas neuronales

Para el desarrollo del programa se siguió la metodología recien descrita, usando como error E al error cuadrático medio, y al sigmoide como función f de activación. Respectivamente:

$$E(Y, y) = \frac{1}{2}|Y - y|^2$$
$$f(t) = \frac{1}{1 + e^{-t}}$$

Cabe destacar que $||$ es la norma euclidiana común, y que como se trabaja con muchos datos a la vez, se usa el promedio del error cuadratico medio:

$$< E(Y, y) >= \frac{1}{2n} \sum_{i}^{n} |Y_i - y_i|^2$$

Se creó una clase de capas de redes neuronales, con miembros privados a la cantidad de inputs (neuronas de la capa anterior o dimensión de los datos de entrada iniciales), cantidad de outputs (neuronas en la capa), matriz de pesos, vector de constantes, y punteros a la función de activación y su derivada. Tanto los pesos como las constantes se inicializan con valores double random entre -1 y 1.

Las funciones internas permiten usar las matrices o funciones de activación como se requiera, en general son todas intuitivas.

Las funciones externas permiten una multitud de cosas. Destacando las más importantes:

1) Red, que crea una matriz de capas neuronales (red neuronal).

2) Forward, que calcula con la red sin entrenar.

3) Train, que calcula y luego entrena a la red (solo una iteracion forward y una backward).

4) Entrena_guarda, que realiza todo lo que implica entrenar a una red por una cantidad arbitraria de iteraciones, guardando datos relevantes como pruebas de la red y error en el tiempo.

5) Entrena_guarda_overf, lo mismo que la anterior pero con las consideraciones para evitar overfitting.

3.2. Principales características y uso

Me especialicé en funciones en 3-D, de manera que se tienen inputs de dos dimensiones y outputs en una. Las funciones que hacen los detalles del proyecto están enfocadas a estos casos, pero la clase incluyendo las funciones de entrenamiento son absolutamente generales.

El main del proyecto recibe como argumentos de main en orden, datos en x (inputs iniciales), datos en y (outputs esperados), y ratio de aprendizaje. Claramente datos en x y en y deben tener la misma cantidad de filas.

Se crea una red neuronal según una estructura en el archivo (hay que cambiarla a mano si se desea otra, en el vector de enteros "estruc" linea 15). Se pregunta si se desea estudiar overfitting, de escoger si, se seleccionan de forma aleatoria un 30 % de los datos en x e y (manteniendo correlación) para solo probar la red y se usa el otro 70 % para entrenarla. Para ambos casos, se pregunta tambien cuantas iteraciones se desean, y si se desean chequear los datos. Esta última opción permite probar la red con una cantidad de puntos equiespaceados del plano cartesiano, con límites un poco más extensos que los entregados en datos en x (esta característica es específica para las funciones 3-D), y guardar los resultados para poder visualizar una evolución de la red. De escogerse también se preguntará cada cuantas iteraciones se hará el chequeo.

Luego de escoger todo, se usa entrena_guarda o entrena_guarda_overf para calcular y guardar todo, dependiendo si se decidio estudiar overfitting o no. Se presenta además en pantalla la iteración y el o los errores correspondientes a esta. Al terminar todas las iteraciones se permite guardar o no la red entrenada.

Es posible también añadir un argumento de main extra, que sea una red en el mismo formato en que se guardan, de manera de poder entrenar a la misma red en múltiples ocasiones para refinar esta.

Cabe destacar que las redes funcionales funcionan mucho mejor con datos normalizados entre los límites de la función de activación, por lo que se hace un estudio interno y de ser necesario se normaliza al calcular y desnormaliza al entregar los resultados, de manera que se puedan usar las redes sin preocuparse de los mínimos o máximos de los datos.

Todo lo creado se guarda en el mismo directorio en una carpeta llamada "evolucion_redi", donde i es cualquier número desde 0. El programa ve en la carpeta actual y de existir ya el 0, se crea el 1, y así iterativamente, de manera que nunca se sobreescriba una carpeta con resultados. De querer cambiar el nombre, se puede hacer en la linea 25.

Los resultados en las carpetas creadas son; tablas del error en función del tiempo, tabla de los datos x e y pegados uno al otro (para facilitar visualización), si se escogió overfitting tabla del error de training e intentos, y si se escogió "chequear" las redes, tablas con los resultados de estos chequeos como "Estado_i" siendo i la iteración correspondiente (cabe destacar que los estados se crean con la cantidad de ceros necesarias para que al listar salgan en orden). Si se escogiar guardar la red, se guarda en esta misma carpeta con el nombre de "red.nn".

Existe en el tar un script de bash que recibe como argumento de main una de las carpetas con resultados. Este se encarga de reemplazar todos los Estados por un gif que muestra la evolución de estos y además generar imagenes de la evolución del o los errores. El gif presenta el chequeo de la red como una sabana, los datos entregados como puntos fijos, y el archivo del cual proviene cada imagen (con lo que se puede saber en que iteración se tenía ese estado). El script de bash además recibe como argumentos opcionales los límites del ploteo para 'x', 'y' y 'z'.

Por último, se incluye una carpeta adicional en el comprimido llamada crea_datos, esta solo facilita la evalución de funciones y creación de archivos con inputs y outputs que pueda recibir el programa de redes neuronales.

4. Ejemplos y resultados

Se incluye una carpeta de ejemplos con tres directorios dentro. De más simple a más complejo son:

1) Dos circulos: Sistema simple de dos circunferencias concéntricas con cierta cantidad de ruido (no perfectas) con valores binarios (la externa 0, la interna 1). Se incluyen los datos en x e y ademas de dos carpetas de resultado, una considerando overfitting y la otra no.

2) Paraboloide: Paraboloide común de ecuación $z = x^2 + y^2$, se incluyen igualmente los datos en x e y más cuatro carpetas. Una es el estudio con overfitting, y las otras tres estudio sin overfitting.

   Estas últimas son para exponer uno de los problemas al trabajar con un ratio de aprendizaje fijo: el de caer en espacio de cuasi-soluciones oscilantes, pues el mínimo podría estar a una distancia menor que la del paso que estoy dando. En el resultado_noverf presenta un caso en que llega al estado oscilante casi al final, cuando tiene bien lograda la forma del paraboloide, la segunda carpeta (con un 2), muestra un caso con ratio de aprendizaje menor, de manera que nunca llega al estado oscilante (al menos en esa cantidad de iteraciones), y el tercero muestra un ejemplo con un ratio mucho mas grande, de manera que empieza a oscilar muy pronto.

3) Bessel_superficie: Una lámina donde la componente radial es una función de bessel de primera especie y orden 0. Los datos en x e y se generaron usando una función de la tarea 2 de este curso (ambos presentes en la carpeta).

   Además hay cinco carpetas que muestran la evolución de una única red. Resultado1 es el comienzo, se realizaron muchas más iteraciones en comparación a los otros sistemas. Las otras cuatro están para denotar otro problema de las redes, el como cuesta aproximarse a soluciones exactas cuando el error ha bajado demasiado. Se muestra la evolución de la misma red, donde el error siempre baja, pero poquísimo, de manera que la evolución se ve fuertemente estancada. En este caso, creo que el problema se origina por una mala distribución de datos, de manera que un mayor porcentaje de estos ya se acomodaron bien por parte de la red, y pocos quedan mal ajustados, lo que pesa poco en el promedio del error (usado para ajustar los valores de esta).

   Se intentó con más puntos o estructuras de red más complejas pero el tiempo de cómputo aumentaba significativamente

A modo de conclusión, fue muy estimulante el desarrollo de este proyecto. Si bien sus aplicaciones a la física están muy acotadas, no deja de ser un sistema de algoritmos de gran utilidad para mucho de los procesos en que nos vemos involucrados hoy en día. Cabe destacar que para aprovechar profundamente su potencial se requiere de mucha más refinación en la estructura de las redes, y gran poder de cómputo en paralelo (como usando CUDA) con cantidades masivas de datos.