Tarea 1 - Entrenar una red neuronal

Profesor: Alexandre Bergel
Auxiliar: Juan-Pablo Silva
Ayudantes: Marco Caballero
María José Berger

Para esta tarea deberán hacer uso de la implementación de la red neuronal que han estado programando en clases. Este enunciado detalla lo que se espera que la red neuronal pueda hacer, qué debe tener y las evaluaciones que deben realizar.

1 Contenidos

En esta sección explicaremos lo que deberán hacer en la tarea y lo que se espera que tengan implementado.

1.1 Neuronas

Como hemos visto durante las clases, las neuronas son la base de toda red neuronal. Estas poseen pesos y un sesgo (*bias*), que se utilizan para tomar decisiones.

La tarea espera que sus implementaciones de neuronas puedan hacer lo siguiente:

- **Inicialización**: su neurona puede crearse y funciona para un número variable de argumentos. Con argumentos variables nos referimos a que el número de pesos asociados al input de la neurona no esta predefinido, sino que cada neurona puede definir su propio número de pesos.
- **Predicción**: su neurona debe tener un método (comúnmente llamado `feed`) que reciba una lista de argumentos. Estos parámetros deben coincidir con los pesos definidos en la neurona. El método debe evaluar el input con respecto a los pesos y el *bias* y retornar el valor resultante.
- **Función de activación**: su neurona debe poseer una función de activación. Recomendamos que esta función sea una variable dentro de la inicialización de la neurona, y se guarde como propiedad. La idea es que no todas las neuronas estén obligadas a usar la misma función. Entre las funciones que conocemos, se deben permitir `step` (la función escalón), `sigmoid` (la regresión logística), y la tangente hiperbólica `tanh`.
- **Aprendizaje**: su neurona debe ser capaz de aprender basada en un conjunto de entrada. Comúnmente el método que permite esto se llama `train` y recibe 2 argumentos, el primero corresponde a la entrada para la neurona, y el segundo corresponde al valor de salida esperado que debería dar la neurona.
- **Robustez**: su neurona debe ser robusta en el sentido que no debe permitir *inputs* que no estén alineados con lo declarado inicialmente respecto a los pesos. Además, debe cuidar que

las entradas de su neurona efectivamente sean numéricas.

1.2 Red Neuronal

Una red neuronal solo corresponde a un conjunto de neuronas. Se recomienda estructurar esta parte como que una capa o *layer* es un conjunto de neuronas, y una red es un conjunto de capas. Su red neuronal debe ser capaz de hacer las siguientes acciones:

- **Iniciar red**: su red debe poder crearse basado en 4 argumentos: número de capas (un valor entero), número de neuronas por capa (una lista de valores enteros), número de entradas (un valor entero), y el número de valores de salida (un valor entero). Adicionalmente, puede ser en el constructor o como un método separado, configurar las funciones de activación que se quieren usar para cada capa, configurar la tasa de aprendizaje (*learning rate*), y cargar pesos ya predefinidos.
- **Hacer predicciones**: esto mediante un método `feed` que reciba el input de la red, evalúe todas sus capas en orden, y finalmente tenga un valor de salida.
- **Aprendizaje**: su red debe poder aprender usando un método `train`, que reciba la entrada para la red y el valor resultante esperado. Adicionalmente puede hacer otro método que reciba una lista de entradas, para entrenar sobre un *dataset* completo, pero esto es opcional.
- **Robustez**: su red debe ser robusta en el sentido que no debe permitir *inputs* que no estén alineados con lo declarado inicialmente respecto al número de entradas. Además, debe cuidar que las entradas de su ned efectivamente sean valores numéricos.

1.3 Funciones de activación

Debe implementar las siguientes funciones de activación, las cuales se indican a nivel de red, y luego se propagan a nivel de capa, para cada neurona dentro de la capa. Aquí se indica la función matemática y su derivada.

1.3.1 función `step`

La función escalón viene dada por la siguiente formula:

$$\text{step}(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases} \tag{1}$$

La derivada de esta función es 0 en todos lados (porque es constante), y su derivada en 0 no existe. Otros pueden argumentar que su derivada en 0 es ∞, pero eso no nos sirve a nosotros. Por lo tanto, esta función no sirve para aprender y entrenar una red neuronal, pero debe implementarla por completitud de las primeras actividades.

1.3.2 función `sigmoid`

La función *sigmoid*, o regresión logística, viene dada por la siguiente formula:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

Su derivada es la siguiente:

$$\frac{\partial \text{ sigmoid}(x)}{\partial x} = \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x)) \tag{3}$$

1.3.3 función `tanh`

La función *tanh*, también muy usada, viene dada por la siguiente formula:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4}$$

Su derivada es la siguiente:

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x) \tag{5}$$

1.3.4 Recomendaciones de implementacion

Haga una clase para cada función de activación e implemente métodos para aplicar la función y para aplicar su derivada, por ejemplo para el caso de la función `sigmoid` puede usar lo siguiente:

```
1    import numpy as np
2    class Sigmoid:
3        def apply(self, x):
4            return 1 / (1 + np.exp(-x))
5
6        def derivative(self, x):
7            return self.apply(x) * (1 - self.apply(x))
```

Luego, usando ese código, cuando este propagando los errores en la red mediante las capas, considere el *transferDerivative* como la derivada de la función de activación de la neurona.

1.4 Funciones de error (*loss*)

Aquí es suficiente con que implementen la función de error con la que hemos trabajo todo este tiempo, la función de error cuadrático medio:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{6}$$

Aquí lo que hacemos es restar el valor que tenemos (Y_i) con el valor que queríamos que la red nos diera ($\hat{Y}_i$), luego calculamos el cuadrado de esa resta. Ahora calculamos esto para todos los puntos (ejemplos, datos) que tenemos y calculamos el promedio (sumamos y luego dividimos por el número de puntos sumados).

Use esta formula para mostrar el error de la red. Para entrenar la red usara esta misma función, pero se ve un poco distinta, para ello revise las diapositivas (clase 7), donde se muestra el algoritmo que debe seguir para entrenar la red.

1.5 Dataset

Para la tarea deberá usar un *dataset* un *poco* menos de juguete que compuertas lógicas. Ahora deberá descargar un conjunto de datos de internet, recomendamos el tradicional Iris dataset[1]. Este dataset corresponde a mediciones de las semillas de 3 tipos de flores de iris, que se compone de 7 variables, correspondiendo a características de las semillas, y se clasifica en 3 clases o tipos de flores. La tarea a resolver es, dado ciertas mediciones de la semilla, encontrar a qué flor corresponde. Hay varias distribuciones de este dataset, el que enlazamos contiene 7 variables, pero hay otros que tienen 4.

Para trabajar con el dataset considere lo descrito en las siguientes secciones.

1.5.1 Normalización

Como se vió en clases (clase 6), es necesario normalizar las variables o el modelo (la red) no converge. Para ello considere la siguiente formula:

$$f(x) = \frac{(x - min(x))}{(max(x) - min(x))}(\text{High} - \text{Low}) + \text{Low} \tag{7}$$

Donde *High* y *Low* corresponden a los valores deseados donde se quiere escalar los datos (tradicionalmente *Low* siendo 0 y *High* siendo 1). Las funciones *min* y *max* están actuando sobre el conjunto de datos, pero manteniendo la separación entre variables. Aquí la siguiente ayuda:

[1] `https://archive.ics.uci.edu/ml/datasets/seeds`

	x_1	x_2	x_3	x_4
	4	2	6	11
	6	4	5	109
	2	9	4	64
	6	1	5	42
	9.4	16	5	44
$max(x)$	9.4	16	6	109

La entrada para la función de normalizar puede ser una matriz, en este caso una de 5 filas con 4 columnas. Usando `numpy` puede aplicar la función *max* sobre la matriz a través de un eje, en este caso el primer eje, de la siguiente forma: `np.max(x, axis=0)`. Aplique lo mismo para *min* y luego calcule los datos normalizados.

Hay varias formas de normalizar sus datos. Esta formula solo re-escala los datos para que estén entre 0 y 1, pero no altera la distribución. Es libre de jugar con otras normalizaciones, como centrar los datos a una distribución Gaussiana.

1.5.2 Clasificacion en 1-hot encoding

En el caso de que use el dataset de Iris, veré que estamos clasificando sobre texto. Hay 3 tipos de flores, pero están con nombre. Nuestra red no recibe texto, así que una forma tradicional de arreglar esto es con *1-hot encoding*. Básicamente como tenemos 3 posibles valores, creamos un vector de 3 valores, donde siempre cada posición en el vector representa una de las posibles clases. Entonces, tenemos un vector de puros ceros, salvo alguna de las posiciones donde tenemos un 1.

Como ayuda, considere el siguiente "algoritmo" para obtener la codificación.

1. Cargue los datos desde el archivo.
2. Use un `set` para obtener las clases en las que puede caer su clasificación. Asumiendo que usa *numpy* y que la columna que posee la clase es la última, use `set(datos[:, -1])`.
3. Ahora puede crear un diccionario para ponerle número a sus valores. Enumere los valores del `set` y asigne un número.
4. Recorra sus datos para pasar de texto a número cada una de las clases. Use el diccionario para ir buscando que número corresponde a que clase. Esto terminará con un arreglo de enteros.
5. 2 opciones. Si usa *numpy*, simplemente haga `np.eye(max(arreglo) + 1)[arreglo]`. Esto le dará una matriz, con cada clase como un arreglo de ceros con un 1 en la posición que corresponde. Si no usa *numpy*, cree una matriz de ceros y asigne un 1 en la posición que corresponde a cada número.

6. Ahora "pegue" de vuelta sus datos.

1.5.3 Particion entrenamiento-prueba

Es trampa probar su red con los mismos datos que entrenó, así que debe particionar sus datos, entrenar con una parte y probar con la otra. Los datos con los que prueba no debería mostrárselos a la red. Basta con que seleccione entre un 20 a 30% de los datos exclusivamente para evaluar su desempeño.

1.6 Análisis

Debe producir una matriz de confusión con los datos de evaluación que separo. Asi podrá ver qué tan bien su red aprendió.

También debe generar 2 curvas. Una de aprendizaje de la red, donde por cada época calcula el porcentaje de aciertos de su red, que debería ir subiendo. Y otra en la que se muestre el error por cada época y como este, ojalá, fue bajando.

1.7 Bonus

Puede hacer los siguientes bonus si quiere aprender y experimentar más:

1. Usar `k-fold Cross-Validation`. Una técnica para generalizar y evaluar distintas particiones de los datos, a ver si la red en verdad aprendió o fue pura suerte.
2. Usar múltiples dataset.
3. Variar el número de capas, neuronas, tasas de aprendizaje, etc, y hacer un *heatmap* con las mediciones que haga. Puede comparar qué tan rápido aprende la red, qué tan bien aprende en un número determinado de épocas, u otras apreciaciones que haga.
4. Identificar neuronas que tienen pesos muy bajos y que no están aportando a la red.

2 Evaluación

Para evaluar sus tareas se revisará su código, por lo que se pedirá que las partes relevantes estén comentadas para facilitar la corrección a los ayudantes. Además, para la parte de análisis, debe hacer un `readme` con esta información. A continuación se hace el desglose de evaluación:

- **Código Red** (2.0 puntos): se revisará que se haya implementado lo descrito en las secciones de contenido 1.1, 1.2 y 1.3.
- **Feedback de red** (1.0 puntos): su red debe poder indicar el *error* y *accuracy* por época. Básicamente debe tener alguna forma de indicar qué tan bien la red lo está haciendo por cada

época que pasa. "Qué tan bien lo está haciendo" se define en base a ambos, el porcentaje de aciertos y del error.

- **Dataset** (1.0 puntos): debe utilizar un dataset "de verdad", que no sea compuertas lógicas. Además debe implementar lo descrito en la sección 1.5 sobre normalización y partición de datos (hecho por ustedes o usando alguna librería). También debe agregar la parte de *1-hot encoding* si lo necesita.
- **Análisis** (2.0 puntos): haga una pequeña retrospectiva acerca de su implementación de la red, las dificultades que tuvo y apreciaciones a su eficiencia. Debe, además, presentar y describir las curvas de aprendizaje y error de su red sobre el dataset que utilizó. Todo esto debe ser presentado en el `readme` de su repositorio. **NO** requiere extenderse demasiado, es una descripción de lo que hizo más los resultados que obtuvo. Aquí puede encontrar una guía de como usar *markdown*[2].

3 Entrega

La fecha de entrega de la tarea es el **jueves 29 de agosto**. Como usaremos Github para bajar sus tareas, basta con que en U-cursos suban **cualquier** archivo, y en los comentarios de entrega agreguen el enlace a su repositorio en Github.

Los análisis e imágenes de sus gráficos deben estar en el `readme` de su repositorio, no es necesario hacer un informe ni un documento en *pdf* explicando las cosas. La extensión en corta, solo debe hacer un pequeño reporte en el `readme` con los puntos señalados anteriormente.

[2] `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`