Universidad Tecnológica Centroamericana
Facultad de Ingeniería

CC414 - Sistemas Inteligentes
Docente: Kenny Dávila, PhD

Tarea #4 – Redes Neurales Artificiales (3% Puntos Oro)

El objetivo de esta tarea es la implementación de redes neural artificiales con alimentación hacia adelante (feed-forward). Todas las arquitecturas a probar son totalmente conectadas (fully connected network). Se requiere el uso de la librería Pytorch. En caso de no tener una tarjeta de video (GPU) apropiado, se recomienda utilizar la instalación de Pytorch basada en CPU.

Nota. La Tercera edición del libro de la clase tiene un error en el pseudo código del algoritmo de back-propagation. El error es que la inicialización aleatoria de los pesos aparece dentro del ciclo de entrenamiento (al inicio de cada época). Sin embargo, esta inicialización **debe realizarse solamente una vez al inicio**, antes de la primera época.

Parte 1. Feed-Forward (1.0%)

El objetivo de este punto es la implementación de la estructura básica de una red neural artificial con alimentación hacia adelante. Se debe utilizar la librería PyTorch para crear la arquitectura de dicha red neural. Se debe usar capas totalmente conectadas (fully connected). Es importante notar que Pytorch separa la combinación de las entradas de las funciones de activación. Por eso se debe usar "linear" para generar una combinación de las entradas como se vio en clasa, al resultado se le aplica la función de activación (en este caso sigmoid).

Pytorch provee diferentes mecanismos para inicializar los pesos de la red con valores aleatorios. Debe asegurarse de aplicar alguna de estas estrategias. Pytorch también provee métodos para guardar los pesos de una red neural. Estos son importantes ya que el entrenamiento de una red puede tomar horas o hasta días y es importante guardar y recuperar los pesos que aprendió la red al final de su entrenamiento. Como **requisito** de esta primera parte **debe generar pesos aleatorios con una estrategia valida y debe guardarlos en un archivo** "pesos_parte_1.pkl".

El objetivo de este ejercicio es implementar y validar la alimentación hacia adelante. Para esto, debe seguir los lineamientos y recomendaciones de la librería PyTorch. Es importante notar que esta es una librería Orientada a Objetos, y para que funcionen bien ciertos mecanismos, debe usar herencia de clases de manera apropiada (e.g. su clase RedParte1 debe ser hija de torch.nn.Module). Se debe implementar la función "forward(self, x)" para evaluar la salida de la red. Se recibe como entrada un tensor que representa uno o varios vectores de entrada X y se evalúan las salidas de cada capa de la red hasta producir un tensor de salida y con los vectores de salida de la ultima capa.

Se le pide utilizar su implementación para recrear la arquitectura del ejemplo visto en clase (2 capas con 2 neuronas por cada capa). **En su informe debe reportar las salidas producidas por la red** para los siguientes vectores: <0, 0>, <0, 1>, <1, 0> y <1,1>. Debe reportar resultados usando inicializaciones de pesos aleatorios (incluya los pesos en el reporte).

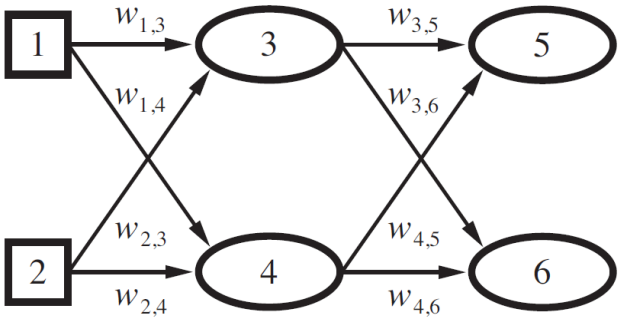

Figura 1. Arquitectura de red neural a Implementar. Los pesos del bias en cada neurona K ($w_{0,k}$) **no** están representados en esta imagen, pero son requeridos en su implementación.

Parte 2. Entrenamiento de Red Neural (1.0%)

En esta parte se le pide utilizar el algoritmo de propagación hacia atrás para entrenamiento de la red neural. PyTorch provee los mecanismos para utilizar dicho método de forma sencilla mediante el optimizador SGD (Stocastic Gradient Descent). Dicho optimizador ayuda a calcular y optimizar la red en base a los gradientes. Debe tener en mente que la red siempre calcula los gradientes por defecto a menos que se use la función forward dentro del entorno “torch.no_grad()”. Es importante limpiar los gradientes despues de entrenar cada lote usando la funcion “zero_grad()” del optimizador.

El objetivo de esta parte es aprender a utilizar data loaders, funciones de perdida y optimizers en PyTorch para entrenar una red neural. En este caso, se utiliza un dataset extremadamente pequeño, asi que se pide usar un Batch size de 2 (solo 2 ejemplos por cada lote de entrenamiento). Como función de perdida (**loss function**), se le pide utilizar el MSELoss. Debe crear su propio “data loader” para los datos provistos (“part2_train_data.csv”). Es importante notar que los datos provistos para esta parte utilizan múltiples columnas (y1, y2) para indicar las salidas que debe aprender cada neurona de salida.

Debe recordar que la inicialización de los pesos se hace solamente una vez al inicio y no dentro de cada ciclo de entrenamiento (época) como se muestra en el libro. El algoritmo de back-propagation se basa en épocas. Por cada época, la red es ajustada usando **todos los datos de entrenamiento** disponibles. En esta versión usamos lotes de tamaño 2. El algoritmo se detiene cuando se han realizado un número máximo de épocas o si se han hecho múltiples épocas continuas en las que el error en datos de validación no disminuye o incluso se incrementa. Al terminar el proceso de entrenamiento, **los pesos aprendidos deben guardarse en un archivo**.

El algoritmo recibe como **entrada** los “datos de entrenamiento” y “máximo número de épocas”. Al final de cada ronda de entrenamiento (época), el algoritmo debe **imprimir estadísticas** como el MSE entre los

vectores de salida esperados (ground truth) y los vectores de salida de la red (output). Recuerde que esta evaluación debe calcularse dentro de un entorno “torch.no_grad()”. Note que es conveniente que se use un **formato de salida** que sea fácil de **tabular,** pero también es válido guardar directamente las estadísticas en un archivo csv para su posterior análisis. En este caso NO hay datos de validación, por lo tanto el proceso simplemente se debe detener cuando se han hecho el numero indicado de épocas. Se recomienda usar un learning rate pequeño (e.g. LR= 0.1).

Se le pide **entrenar 20 redes neurales usando la misma arquitectura simple de la parte 1** y la función suma binaria de dos entradas x1 y x2 usando los siguientes datos de entrenamiento en el archivo “part2_train_data.csv”. Cada red neural deberá ser inicializada con **pesos aleatorios**. Cada red debe ser entrenada por un **total de 100 épocas**. Por cada red y por cada época se obtendrá un valor MSE (20 redes x 100 epochas = 2000 valores MSE). La idea es analizar como evoluciona los valores del MSE por cada época de entrenamiento. Para esto debe agrupar los valores MSE de todas las redes para una misma época. Es decir, los 20 valores MSE de la época 1, los 20 valores MSE de la época 2, y asi sucesivamente. Luego, se le pide generar un gráfico de líneas con **3 líneas** mostrando lo siguiente: valor **máximo** de MSE por época, valor **mínimo** de MSE por época, valor **promedio** de MSE por época. **No hay datos de validación para este ejercicio**.

x_1	x_2	y_3 (carry)	y_4 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Datos de entrenamiento a usar

Parte 3. Clasificación (1.0%)

El objetivo de esta parte es aplicar lo aprendido en las 2 partes anteriores para crear y evaluar múltiples arquitecturas de redes neurales para resolver un problema de clasificación. A diferencia de la arquitectura anterior, la red neural de clasificación **DEBE** tener una **capa extra al final, log softmax**, que es la que se asegura de combinar las salidas de todas las neuronas y producir seudo probabilidades por cada clase. Otro detalle importante es qué para problemas de clasificación, es preferible usar una función de perdida diferente al MSE. En este caso, se le pide usar el **Negative Log Likelihood Loss** (NLLLoss).

El algoritmo de entrenamiento recibe como **entrada** los “datos de entrenamiento”, “máximo número de épocas”, “datos de validación”, “valor épsilon” y “máximo de rondas sin decremento”. Al final de cada ronda de entrenamiento (época), el algoritmo debe **imprimir estadísticas** como el NLL Loss promedio entre los vectores de salida esperados (ground truth) y los vectores de salida de la red (output). También se deben imprimir las mismas estadísticas para los datos de validación al finalizar cada época. Note que es conveniente que se use un **formato de salida** que sea fácil de **tabular,** pero también es válido guardar directamente las estadísticas en un archivo csv para su posterior análisis. Deberá hacer un conteo al final de cada ronda donde verifique si la diferencia entre los valores NLL de validación de la ronda anterior y la ronda actual es menor al valor épsilon. Si se acumula un **total de épocas continuas** igual al valor “máximo

de rondas sin decremento", entonces se detiene el proceso de entrenamiento. El entrenamiento también se detiene de forma automática si se alcanza el máximo número de épocas especificado.

Para este problema, se reutilizarán los datos de la compañía aseguradora de la tarea #3 ("seguros_training_data.csv" y "seguros_testing_data.csv"). Anteriormente se clasificaron dichos datos alcanzando un accuracy del 70% con un solo regresor logístico. Una red neural basada en sigmoides equivale a la combinación de múltiples regresores logísticos, por lo que se espera que su rendimiento sea superior al de un solo regresor.

Se le pide entrenar una serie de redes neurales con el fin de encontrar una arquitectura capaz de obtener buenos resultados en la clasificación. Es importante tomar en cuenta que las redes neurales son sensibles a los rangos de los valores de entrada. Por esta razón se recomienda usar normalización de los datos de entrada. Así mismo, se proveen las etiquetas como cadenas de texto. Es requerido usar codificación y convertir estos valores en vectores binarios ("one-hot encoding"), donde cada clase activa uno y solamente uno de los valores de salida para un total de 2 valores binarios. Al momento de realizar la clasificación usando la salida de la red, se debe tomar la neurona con el valor de salida más grande como la clase correspondiente (basado en la función softmax).

Para cada arquitectura, se le pide entrenar la red usando los datos de entrenamiento provistos en el archivo "seguros_training_data.csv". Se deben entrenar la red por un **máximo** de 200 épocas. También se le proveen datos de validación en el archivo "seguros_validation_data.csv", los cuales deberá combinar con un valor épsilon de 0.05 y un máximo de 3 épocas para determinar si el entrenamiento debe parar antes. Una vez completado el entrenamiento, entonces se le pide usar los datos de prueba en "seguros_testing_data.csv" para evaluar la red. A diferencia de la parte 2, aquí estamos haciendo clasificación y por lo tanto aquí se deben imprimir las siguientes métricas al final de cada época: accuracy (entrenamiento), promedio de F-1 por clase (entrenamiento), accuracy (validación), promedio de F-1 por clase (validación). Al final del entrenamiento, deberá imprimir las siguientes métricas para los datos de prueba: matriz de confusión, accuracy y promedio de F-1 por clase. Se le pide **resumir** las métricas sobre **datos de entrenamiento** y **validación** usando **gráficos de línea** en su **reporte**.

Se le pide probar las siguientes arquitecturas de red:

1. 5 entradas, una capa oculta con 4 neuronas, y una capa de salida con 2 neuronas.
2. 5 entradas, una capa oculta con 8 neuronas, y una capa de salida con 2 neuronas.
3. 5 entradas, dos capas ocultas con 4 neuronas, y una capa de salida con 2 neuronas.
4. 5 entradas, dos capas ocultas con 8 neuronas, y una capa de salida con 2 neuronas.
5. Una arquitectura definida por usted en base a los resultados anteriores.

Debe incluir las métricas solicitadas por cada una de las configuraciones anteriores en su reporte. Adicionalmente, se le pide contestar las siguientes preguntas:

1. ¿Cuál de las 5 arquitecturas de red funciono mejor y por qué?
2. ¿En que se basó para crear su propia arquitectura de red neural?
3. ¿Cuál es la utilidad de los datos de validación?

Se dará **un punto oro extra** por investigar sobre normalización por lotes (batch normalization) y utilizarlo de manera apropiada para entrenar redes neurales con al menos 3 capas ocultas.

Otras políticas

1. Esta tarea deberá trabajarse y entregarse **individual** o en **parejas.**
2. Las expectativas son iguales independientemente si se hace individual o en pareja.
3. La entrega será **un solo archivo comprimido (.zip o .rar)**. Dentro de dicho archivo debe contener su respectivo reporte **en formato PDF**. También debe contener los scripts de Python que se usaron para contestar cada punto.
4. Si se hace en parejas, ambas personas deben subir el mismo archivo.
5. El **plagio** será penalizado de manera severa.
6. Los estudiantes que entreguen una tarea 100% original recibirán una nota parcial a pesar de errores existentes. En cambio, los estudiantes que presenten tareas que contenga material plagiado recibirán 0% automáticamente independientemente de la calidad.
7. Tareas entregadas después de la fecha indicada solamente podrán recibir la mitad de la calificación final. Por esta razón, es posible que **un trabajo incompleto pero entregado a tiempo termine recibiendo mejor calificación que uno completo entregado un minuto tarde**.