Sartenejas, 2 de noviembre de 2009
Universidad Simón Bolívar
Inteligencia Artificial II - CI-5438
María Gabriela Valdes 05-39020
Juan Garcia 05-38207
Sep – Dic 2009

Informe Tarea 1

– Resumen

Parte I:

1. Perceptrón:

El objetivo es implementar un perceptrón de dos entradas, el cual debe entrenarse para que aprenda las funciones booleanas AND, OR y XOR. Se le pasaran en la entrada los ejemplos y se actualizaran los pesos correspondientes, con el fin de obtener la salida esperada para cada ejemplo. Ademas se usara el error existente entre la salida obtenida y la salida esperada como condición de parada para el entrenamiento, y mediante gráficas, concluir acerca del comportamiento del perceptrón a lo largo del entrenamiento con diferentes tasas de aprendizaje.

A través de este aprendizaje se pudo comprobar que solo podemos aprender las funciones AND y OR, dado que son linealmente separables, lo contrario de la XOR, que no pudo ser aprendida por el perceptrón que implementamos. La rapidez en la convergencia del algoritmo según la tasa de aprendizaje es bastante notable. Y el error, como era de esperarse oscilaba a través de las iteraciones, pero siempre convergían a 0 al entregar una salida satisfactoria, a diferencia del XOR, que por su condición, el error convergía a otro numero infinitamente, sin darnos una salida satisfactoria.

2. Regla delta:

Para esta versión, la neurona con dos entradas se entrenara siguiendo la regla delta. De igual manera deberá aprender las funciones booleanas AND, OR y XOR. Probaremos con dos tipos de aprendizaje, incremental (estocástico) y por lotes (estándar) y determinaremos cual de ellos converge mas rápido. Ademas de probar el entrenamiento variando las tasas de aprendizaje, dejándolas constantes y reduciéndolas en función de el numero de iteraciones.

Parte II: Backpropagation

Para esta sección de la tarea dejamos las redes de una sola etapa para agregar una capa

intermedia, la cual permitirá realizar aprendizajes mas complejos y de sistemas no linealmente separables. Vamos a entrenar una una red para que permita reconocer dentro de un área especifica, si un punto dado pertenece al área general o a un circulo incrustado en la misma. Para ello aplicaremos el algoritmo de backpropagation. Tendremos 6 conjuntos de entrenamiento, 3 dados previamente y 3 generados aleatoriamente con los puntos repartidos equivalentemente, y para cada tipo con 500, 1000 y 2000 puntos. Y para probar nuestra red entrenamiento ingresaremos una cantidad de 10000 puntos distribuidos uniformemente por toda el área, y procederemos a graficar el resultado, diferenciando mediante colores la pertenecía de los puntos dados, si al área rectangular o al circulo.

Al realizar el entrenamiento para un numero variable de neuronas intermedias, variando de 2 a 10, se pudo observar que el mismo era mucho mas efectivo para un numero alto de neuronas intermedias, pudiendo calcular la pertenencia de los puntos mas efectivamente.

- **Detalles de Implementación:**

Para implementar cada una de las neuronas y redes se utilizo el lenguaje Python, que aunque se puede considerar mas lento que otros lenguajes de programación, gracias a su facilidad y rapidez para escribir programas, nos permitió hacer un uso mas eficiente del tiempo, ya que la depuración y las pruebas realizadas posterior a desarrollar los códigos fueron bastante extensas.

Parte I:

1. Perceptrón:

Primeramente se implemento una clase “Perceptron”, la cual tenia por campos el vector de ejemplos de entrada, el vector de pesos, el vector de targets o salidas esperadas, y se llevaba cuenta de la salida que iba generando el entrenamiento, el error y el numero de iteraciones que se tardaba en entrenar. Los métodos para esta clase eran, el constructor de la clase “__init__”, que inicializaba el perceptrón con las entradas deseadas y un método de entrenamiento “training”, que corría todo lo necesario para entrenar el perceptrón para un conjunto de ejemplos dado.

Ademas se necesito implementar otras funciones como “prod_punto”, la cual permitía realizar el producto escalar entre dos vectores, necesario al momento de calcular la salida del perceptrón en cada iteración. Una función “evaluation” que dado el perceptrón entrenado y un conjunto de ejemplos, calculaba la salida para cada entrada, y así nos permitía conocer si el entrenamiento había sido efectivo, y el conjunto de pesos era el correcto para una función dada.

El programa principal es un ciclo que itera sobre las diferentes tasas de entrenamiento dadas, y que para cada función a probar crea un perceptrón con la entrada y targets necesarios, y que luego de entrenar el perceptrón, evalúa la función, se obtiene la salida y se grafica el error en función del numero de iteraciones.

NOTA: en el código entregado la parte que grafica se encontrara comentada, mas se anexan las gráficas obtenidas para cada tasa de aprendizaje en cada una de las funciones aprendidas.

2. Regla delta:

La clase principal implementada es “LinerUnit”, que representa una neurona artificial de dos entradas. Se encuentra definida por una serie de campos: los inputs o las entradas a la neurona, el numero de ejemplos en la entrada, el vector de pesos, cantidad de arcos de peso, un vector de targets, un vector de salidas, se lleva el registro del error y del numero de iteraciones.

Ademas se definen una serie de métodos para esta clase, que permitirán el desarrollo y ejecución del algoritmo de entrenamiento con la regla delta. Se define el método “update_weights” que actualiza los pesos en cada iteración en el por lotes, “incremental_update_weights” que actualiza los pesos para el incremental, “compute_output” que calcula la salida de la neurona, “compute_error” que calcula el error en el por lotes, “compute_incremental_error” que calcula el error en el incremental, “delta_rule_training” que es la función principal para el entrenamiento en el por lotes y “incremental_delta_rule_training” que es la función de entrenamiento para el incremental.

Parte II: Backpropagation

La clase principal para implementar este algoritmo es la de “SigmoidUnit” que representa una unidad sigmoidal, ya sea de entrada, intermedia o de salida. Los campos que definen esa clase son, el tipo de la unidad, si es de entrada, intermedia o salida, un vector de entradas, un vector de salidas, un vector de pesos, la cantidad de ejemplos en la entrada, el error que se genera y un umbral que se utiliza al momento de calcular las salidas de las unidades.

Ademas, esta clase define una serie de métodos que usara el algoritmo para su ejecución El método “compute_output” que calcula las salidas para cada unidad, “compute_error_out” que calcula el error para las unidades de salida, “compute_error_hidden” que calcula el error para cada una de las unidades intermedias, “update_weights” que en cada iteración actualiza los pesos de la red, después de que se hayan calculados los errores, y por ultimo una función “print_weights” que nos ayudada imprimiendo los pesos que se iban calculando.

Dado que los conjuntos de entrada eran, por una parte provenientes de un archivo, y por otra generada aleatoriamente, tenemos dos funciones que nos ayudan a realizar dichas actividades, “read_patterns” que lee del archivo los puntos y targets y los entrega en un vector de pares y un vector respectivamente, y “generate_patterns” que genera dos vectores, el de pares de puntos y el de sus respectivos targets.

La función principal sera la de “backpropagation” que es donde se aplica específicamente el algoritmo, y que hace uso de la clase “SigmoidUnit”, sus campos y sus métodos para ir entrenando la red y entregar una red con pesos actualizados de manera que evalúen nuestro conjunto de prueba de manera efectiva. Para generar este conjunto de prueba usamos la función “generate_test_set” que devuelve un vector con 10000 puntos distribuidos uniformemente sobre el área delimitada.

Por ultimo, tenemos la función que grafica los puntos del conjunto de prueba en pantalla, y que se llama “plot_results”. Simplemente toma la salida de la red, y dependiendo de como se haya calculado la pertenencia de los puntos, la pintara de azul en caso de pertenecer colo al rectángulo y de rojo en caso de pertenecer al circulo.

- **Presentación y discusión de resultados:**

Parte I:

1. Perceptrón:

Los resultados para el entrenamiento del perceptrón para las funciones AND y OR fueron muy satisfactorias. Dado que las mismas son funciones linealmente separables, el perceptrón pudo ser entrenado para evaluar dichas funciones correctamente. Los detalles mas relevantes observados al momento del entrenamiento y del la evaluación, son con respecto a la variación de la tasa de aprendizaje y el error.

Para tasas de aprendizaje bajo, el perceptrón realizaba un numero considerable mayor de iteraciones para lograr los pesos ideales, es to es, que como la tasa era baja, el producto escalar entre las entradas y el vector de pesos tardaba mas en cruzar el umbral que determinaba si una salida era un cero (0) o un uno (1). Por lo tanto para tasas de aprendizaje altas, el numero de iteraciones disminuía ya que ese umbral era sobrepasado mas rápidamente

Una observación importante con respecto al error, era que este representa la condición de parada del algoritmo de entrenamiento, ya que cuando era alcanzado un vector de pesos que validaba cada uno de los ejemplos de la entrada, el error convergía a cero (0), y en este punto se podía concluir que el perceptrón estaba entrenado. Esto ocurría así para los dos funciones, el error en cada iteración oscilaba entre valores dentro de un rango predeterminado, pero al final siempre debía converger a cero (0).

Para el caso de la función XOR, la cual no es linealmente separable, el entrenamiento de perceptrón, como era de esperarse, fue un fracaso. Al evaluar la el perceptrón con la entrada especifica nunca se logro obtener una salida correcta. Esto se evidencia mas claramente en el comportamiento del error, el cual nunca convergía a cero, sino que mas bien aumentaba y permanecía allí por mucho tiempo, teniendo así que establecer una condición de parada mas rigurosa, basada en un numero de iteraciones fijo de 1000. Esto demuestra que el entrenamiento para funciones no linealmente separables no puede realizarse mediante un perceptrón.

Anexo se presentan las gráficas de error obtenidas para cada función, y con la variación de tasa de aprendizaje correspondiente.

2. Regla delta:

El entrenamiento de las neurona para las 3 funcionas fue satisfactoria, sin importar si las funciones eran o no linealmente separables. Probamos para los dos casos de la regla delta, el por lotes e incremental y con la tasa de aprendizaje constante y decreciente.

En general, para la función AND, tanto para el por lotes como para el incremental, para tasas de aprendizaje menor a 0.5 no existe mucha diferencia entre el comportamiento del error para la tasa constante o decreciente. Pero cuando usamos la tasa de 0.99 constante en función de las iteraciones, el error nunca converge, en cambio aumenta con el numero de iteraciones, por lo tanto no logra converger y dar un resultado deseado.

Para la función OR sucede parecido a la función AND, respecto a que para la mayoría de las tasas el comportamiento es similar para la tasa constante o decremental. Sin embargo para las tasas 0.4, 0.5 0.99 constantes durante las iteraciones, el error no converge, en cambio aumenta significativamente, no pudiendo lograr un resultado esperado.

Por ultimo para la función XOR sucede igual que para la función AND, para la mayoría de las tasas se comportan de manera muy parecida, pero la la tasa 0.99 de manera constante, llega un punto en que no converge y mas bien el error crece demasiado y no se obtiene el resultado esperado.

Al realizar corridas con tres tasas de aprendizaje, 0.01, 0.1 y 0.5, y verificar la velocidad y el numero de actualizaciones de pesos, para 0.01, por lotes logra converger mas rápidamente, aunque no con una diferencia tan significativa. Para la tasa de 0.1 los dos algoritmos logran converger casi al mismo tiempo y para tasa 0.5 el algoritmo de incremental logra converger mas rápido para las funciones AND y OR, cuando para la XOR no lo logra, sino que el por lotes converge mas rápido En general podemos decir que la velocidad de convergencia depende significativamente de la tasa de aprendizaje que usemos y de la función que estemos verificando.

Parte II: Backpropagation

Como sabemos, para entrenar la red mediante el algoritmo de backpropagation usamos 6 conjuntos de puntos uniformemente distribuidos en el área, 3 de de ellos dados y 3 generados aleatoriamente y con puntos repartidos equivalentemente entre cada área El proceso de entrenamiento se realizo variando tres factores principalmente.

Primeramente se fue variando el conjunto de entrenamiento, ya que para una cantidad diferente de puntos, y una distribución diferente, la red generada a la salida cambia radicalmente. En segundo lugar se iba variando el numero de unidades intermedias, comenzando con 2 unidades, hasta 10. la variación de neuronas modificaba el resultado enormemente. Pudimos observar que mientras mas neuronas se usaban en la capa intermedia, la red generada lograba mapear el conjunto de prueba de mejor manera. Y por ultimo, cambiar la tasa de aprendizaje también afectaba la red generada, ya que esta es un factor que afecta directamente la actualización de los pesos en cada iteración, e igual que en el perceptrón, permite alcanzar mas rápido un umbral que definirá la correspondencia a un área o a otra.

Para cada una de estas modificaciones siempre se evaluó la red generada con el conjunto de prueba, que eran los 10000 puntos generados uniformemente a lo largo del área del rectángulo Los mejores resultados obtenidos son los siguientes:

- Para el conjunto de prueba dado, de cantidades desiguales:

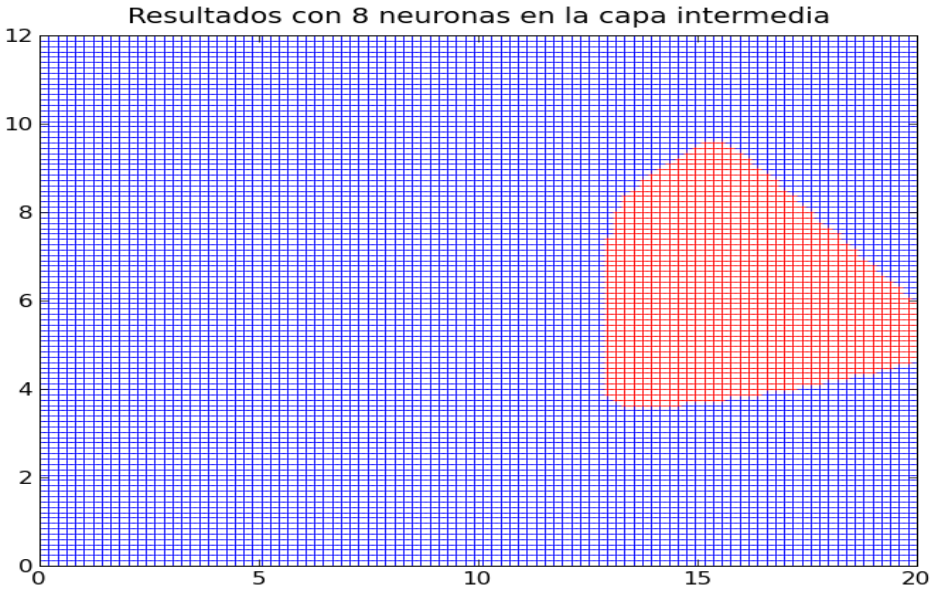
    1. Con 500 puntos, 8 neuronas intermedias y tasa de aprendizaje = 0.1
    2. Con 1000 puntos, 10 neuronas intermedias y tasa de aprendizaje = 0.01
    3. Con 2000 puntos, 9 neuronas intermedias y tasa de aprendizaje = 0.01

- Para el conjunto de prueba generado, de cantidades iguales:

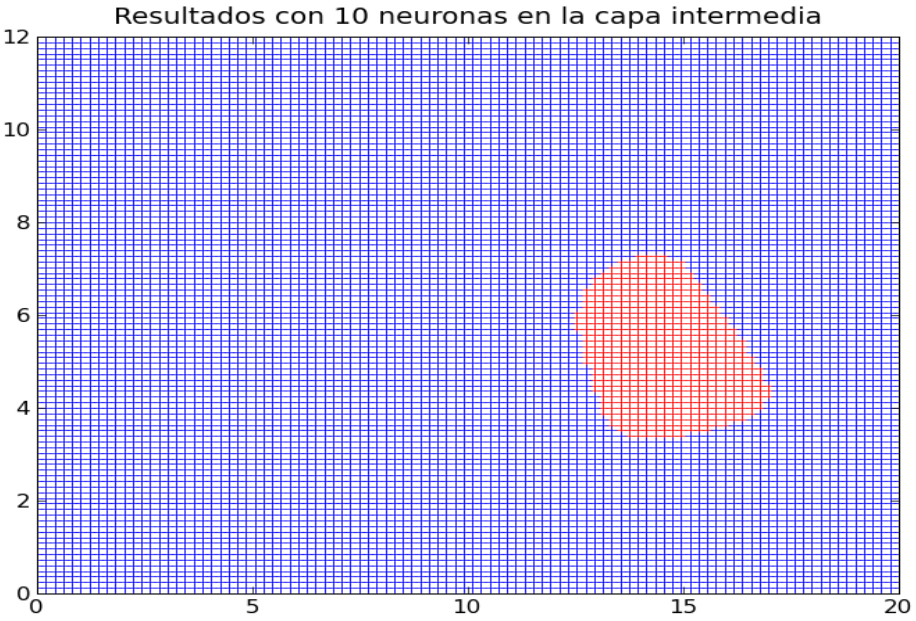
    1. Con 500 puntos, 10 neuronas y tasa de aprendizaje = 0.01
    2. Con 1000 puntos, 7 neuronas y tasa de aprendizaje = 0.01
    3. Con 2000 puntos, 9 neuronas y tasa de aprendizaje = 0.01

Gráficas de Backpropagation

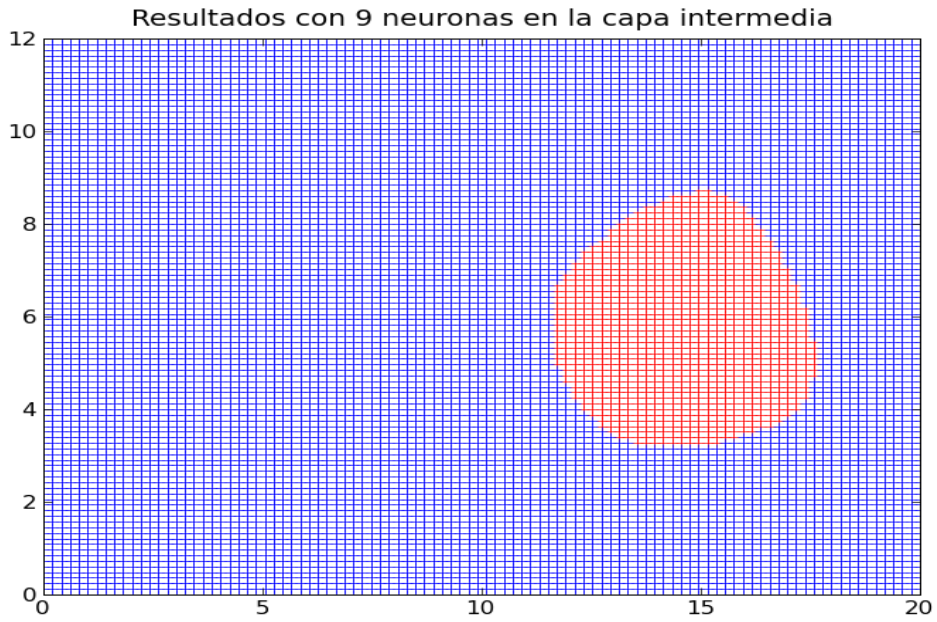
Con 500 puntos, 8 neuronas intermedias y tasa de aprendizaje = 0.1



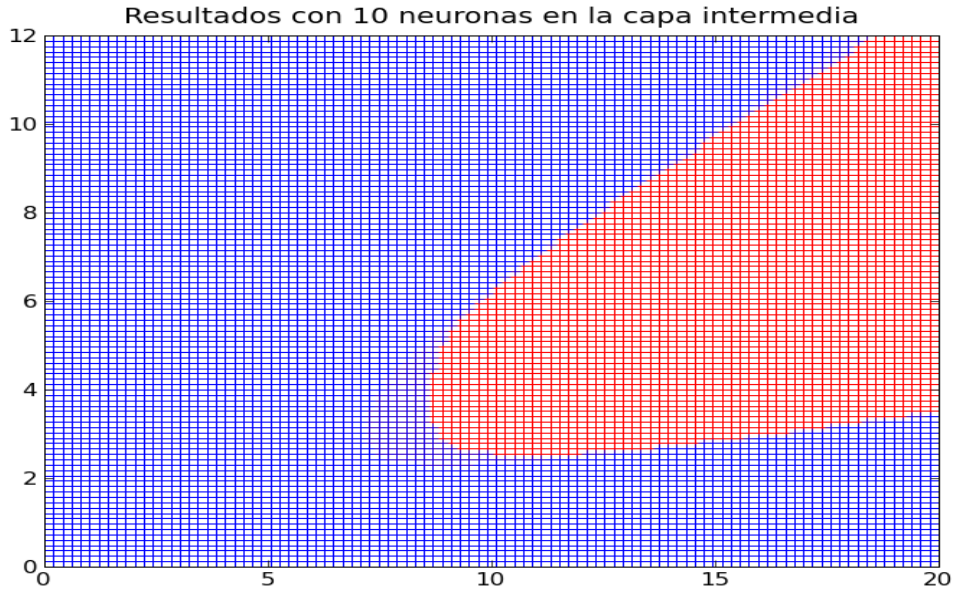
Con 1000 puntos, 10 neuronas intermedias y tasa de aprendizaje = 0.01



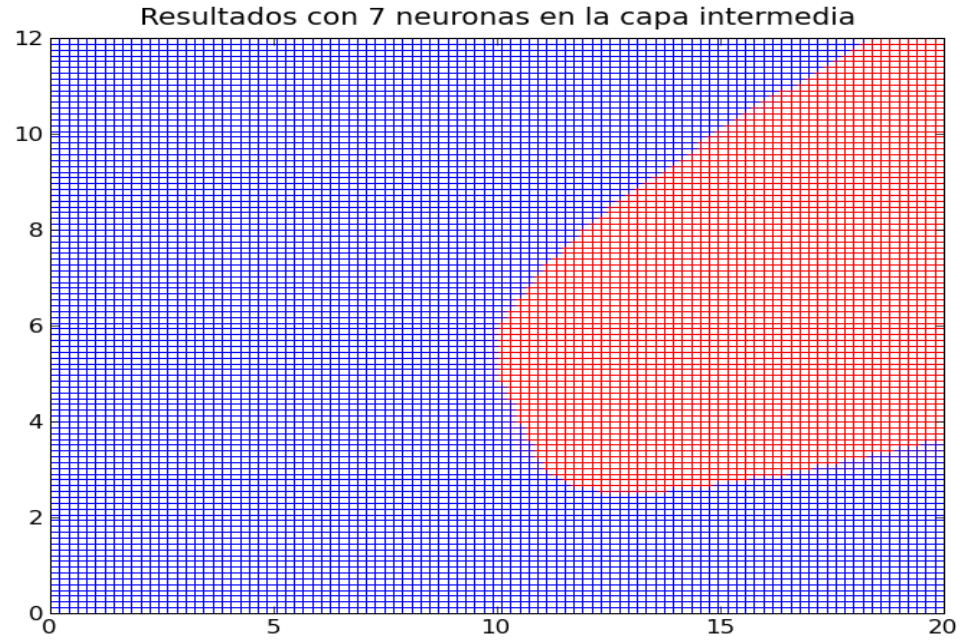
Con 2000 puntos, 9 neuronas intermedias y tasa de aprendizaje = 0.01



Con 500 puntos, 10 neuronas y tasa de aprendizaje = 0.01



Con 1000 puntos, 7 neuronas y tasa de aprendizaje = 0.01



Con 2000 puntos, 9 neuronas y tasa de aprendizaje = 0.01