Tarea 13

Programa. Red Neuronal Multicapa

0.1 Metodología

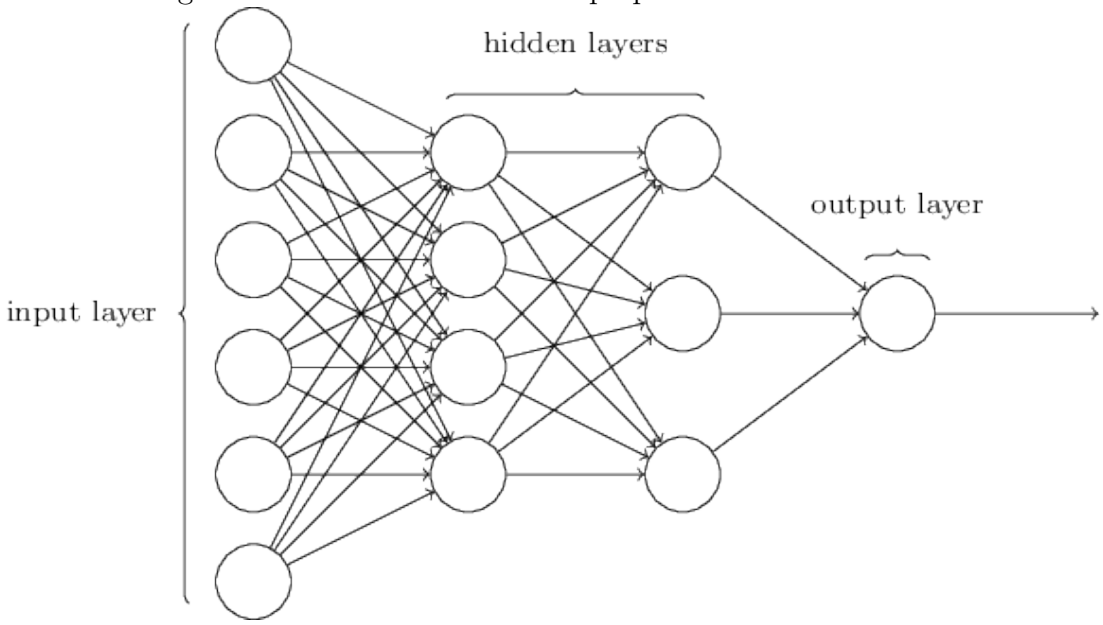
La red neuronal multicapa o perceptrón multicapa, como su nombre lo indica, es una familia de redes neuronales compuestas por capas de perceptron y tienen al menos tres capas, la capa que corresponde a los datos, una capa oculta y la capa de salida donde se obtendrá la predicción.

Para entrenar la red neuronal existen varios métodos como gradiente descendiente, gradiente conjugado, algoritmos genéticos, método Newton o Quasi-Newton.

En el caso de clasificación binaria, la capa de salida tendrá una sola neurona, con una función de activación que de un resultado en el intervalo $[0, 1]$.

En la siguiente imagen se muestra un ejemplo de la arquitectura que se implementará.

Figure 1: Red Neuronal Multicapa para clasificación binaria



Para actualizar los pesos, la idea es tratar de minimizar una función de costo $C(y)$ que nos da una idea de que tan bien o que tan mal es la predicción de la red neuronal. Como ya se comentó, el algoritmo de gradiente descendiente busca minimizar dicha función de costo calculando el gradient del costo respecto a cada uno de los pesos y actualizarlos de acuerdo a dicho gradiente.

Sea W^l la matriz de pesos de la capa l de la red neuronal, b^l el vector de sesgo de la capa l, $z^l = W^l * a^{l-1} + b^l$ la suma ponderada de la capa l y $a^l = g(z^l)$ la salida de la capa l, activada

por una función g.

Se tratará de minimizar una función de costo $C(\widehat{y}, y)$ donde $\widehat{y}$ es la predicción de la red neuronal y donde y es el valor que se quiere predecir correctamente.

Se buscará actualizar los pesos W^l y los sesgos b^l de la siguiente manera:

$$W^l = W^l - \eta \times \frac{\partial C}{\partial W^l}$$

$$b^l = b^l - \eta \times \frac{\partial C}{\partial b^l} \tag{1}$$

Para encontrar las derivadas parciales $\frac{\partial C}{\partial W^l}$ y $\frac{\partial C}{\partial b^l}$ se usa la regla de la cadena:

$$\frac{\partial C}{\partial W^l} = \frac{\partial C}{\partial z^l} \times \frac{\partial z^l}{\partial W^l} \tag{2}$$

Con $\frac{\partial z^l}{\partial W^l} = a^{l-1}$. Sea $\delta^l = \frac{\partial C}{\partial z^l}$.

Aplicando la regla de la cadena para δ^l de la última capa, es decir, con $l = L$, se tiene que:

$$\delta^L = \frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L} \times \frac{\partial a^L}{\partial z^L} \tag{3}$$

Con $\frac{\partial a^L}{\partial z^L} = g'(z^L)$ y donde $\frac{\partial C}{\partial a^L}$ no es más que la derivada de la función de costo respecto a la predicción generada por la red. Con lo que la actualización de la útima capa queda como:

$$\frac{\partial C}{\partial W^l} = a^{l-1} \frac{\partial C}{\partial a^L} g'(z^L) \tag{4}$$

Para las capas restantes lo único que cambiará es el término δ^l:

$$\delta^l = \frac{\partial C}{\partial z^{l+1}} \times \frac{\partial z^{l+1}}{\partial a^l} \times \frac{\partial a^l}{\partial z^l} \tag{5}$$

Como se puede observar, $\frac{\partial C}{\partial z^{l+1}}$ no es más que δ^{l+1} por lo que δ^l se simplifica a:

$$\delta^l = \frac{\partial z^{l+1}}{\partial a^l} \times \delta^{l+1} \times \frac{\partial a^l}{\partial z^l} \tag{6}$$

Con $\frac{\partial z^{l+1}}{\partial a^l} = W^{l+1}$ y con $\frac{\partial a^L}{\partial z^L} = g'(z^L)$.

Y así, el gradiente de los pesos para las capas ocultas queda como:

$$\frac{\partial C}{\partial W^l} = \{(W^{l+1^T} * \delta^{l+1}) * g'(z^l)\} * a^{l-1^T} \tag{7}$$

Mientras que el gradiente del término de sesgo queda simplemente como:

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{8}$$

0.2 Implementación

Para la implementación, se consideró la versión del gradiente descendiente denominada mini-batch, donde en vez de actualizar los pesos obteniendo el gradiente por cada ejemplo de entrenamiento, se toma el promedio de los gradientes para un cierto número de ejemplos que se especifica antes de iniciar el entrenamiento.

Además se aplica un término de momento a la hora de realizar el descenso de gradiente, por lo que se usan las siguientes fórmulas:

$$W^l = W^l - \eta \times V_t$$

$$V_t = \beta \times V_{t-1} + (1 - \beta) \times \frac{\partial C}{\partial W^l}$$

Donde, como ya se mencionó, $\frac{\partial C}{\partial W^l}$ es en realidad el promedio del gradiente para un cierto número de ejemplos. Y donde $\beta \epsilon (0, 1)$.

En cuanto a la función de costo se optó por usar cross-entropy, esto es, $C(y, \widehat{y}) = -(ylog(\widehat{y}) + (1 - y)log(1 - \widehat{y}))$.

En cuanto a la implementación usando POO, se diseñaron las siguientes clases:

1. Clase Neuron: En esta clase es donde se lleva a cabo el calculo de la salida de cada neurona y donde se lleva a cabo la actualización de los pesos.
2. Clase Layer: Clase que contiene a su conjunto de neuronas en un vector y se encarga de que cada neurona produzca su salida y actulicen sus pesos.

3. Clase Network: Clase conformada principalmente por un vector de objetos Layer, encargada de producir la predicción $\widehat{y}$ y hacer que cada capa actualice sus pesos.

4. Clase Activation: Clase que contiene las diferentes funciones de activación para las neuronas. Para esta tarea sólo se implementa la función relu y sigmoid.

5. Clase Data: Clase con funciones específicas para el tratamiento del dataset. Para esta tarea sólo se implementa la función de leer data set.

6. Clase Graph: Clase cuya única función es generar la gráfica de costo vs epochs.

Para crear una red neuronal se realizan los siguientes pasos:

Algorithm 1: Creación red neuronal

```
// Creaci n de objeto red neuronal
Network model = Network(learning_rate);

// Capa de entrada, s lo especifica el tama o de los datos de entrada
model.addData(4);

// Se agregan capas ocultas con tama o y funci n de activaci n arbitraria
model.addDense(5, "relu");

// Se agrega capa de salida
model.addOutput(1, "sigmoid");
```

Para el reporte se muestran pruebas con la arquitectura mostrada en el código anterior, esto es, una sóla capa oculta con 5 neuronas y función de activación relu. De entrada se tiene que los datos forman un vector de dimensión 4 y debido a que se tiene un problema de clasificación binaria, en la salida sólo se tiene una neurona con función de activación sigmoid, ya que da un valor real en intervalo $(0, 1)$.

Una vez que se tiene definida la aquitectura de la red neuronal se puede llevar a cabo el entrenamiento y evaluación del modelo con el data set.

Algorithm 2: Entrenamiento y evaluación

```
int num_epochs = 6000;
int batch_size = 40;

model.fit(x_train, y_train, num_epochs, batch_size);
```

```
model.eval(x_test, y_test);

cout << "\n\nTest Accuracy: " << model.getAccuracy() << "\n" << endl;
```

Después, se grafica el comportamiento de la función de costo conforme avanzan las épocas y se guarda la arquitectura y los pesos de la red neuronal.

Algorithm 3: Gráfica y guardado de modelo

```
vector<double> trainingLoss = model.getTrainingLoss();

plot(trainingLoss);

model.save();
```

Finalmente, se muestra como cargar un modelo ya entrenado previamente.

Algorithm 4: Carga modelo

```
Network model = Network(0);

model.load();

vector<double> y_hat = model.predict(x[0]);

model.eval(x, y);
```

0.3 Resultados

Para el entrenamiento se usó un 80% de los datos para el entrenamiento y un 20% para test con el que se calcula el accuracy. Se fija un learning rate de 0.0001, se realiza un entrenamiento de 6000 épocas, un tamaño de batch de 40 ejemplos y se varía el coeficiente de momento β con tres valores: 0.0 y 0.3 y 0.9.

Para $\beta = 0.3$ y $\beta = 0.9$ se obtiene un accuracy del 100%. Mientras que para $\beta = 0.0$ se obtiene un accuracy del 95%.

Tarea 13

Figure 2: Gráfica costo/pérdida con $\beta = 0.0$

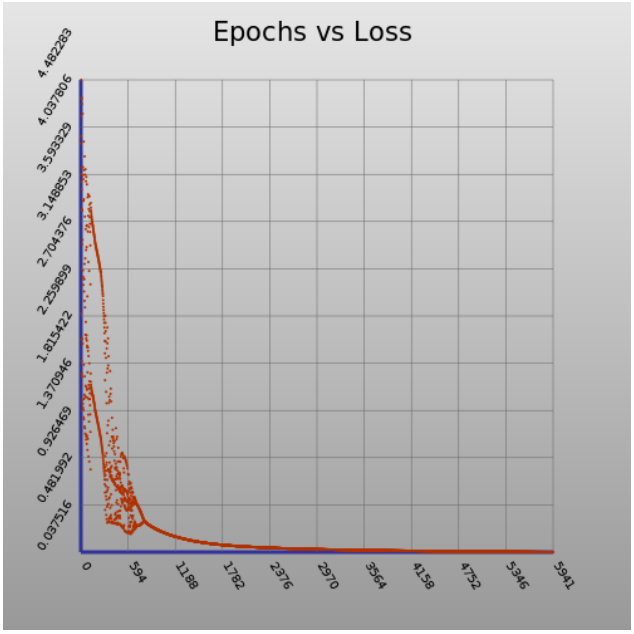


Figure 3: Gráfica costo/pérdida con $\beta = 0.3$

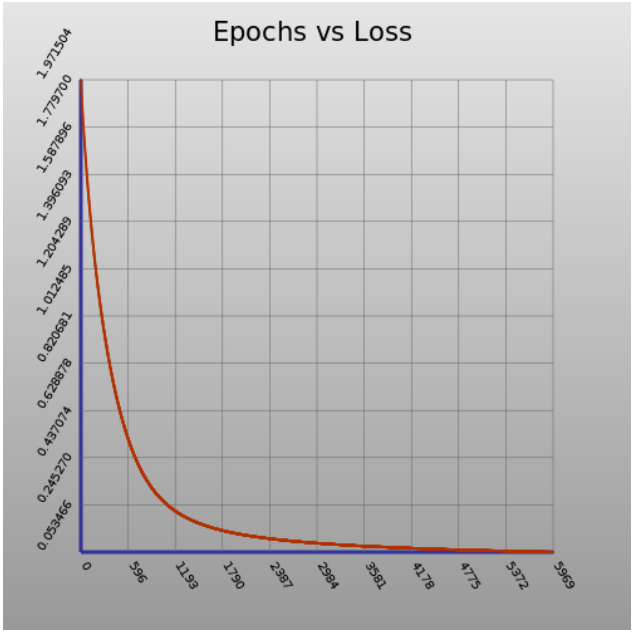
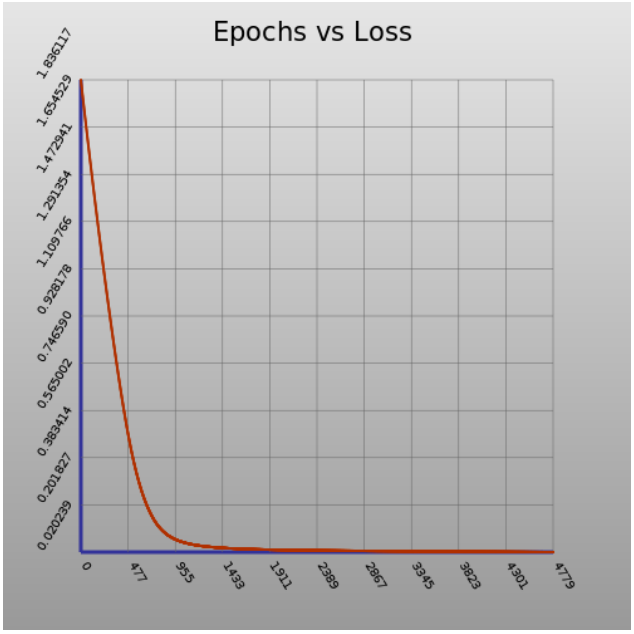
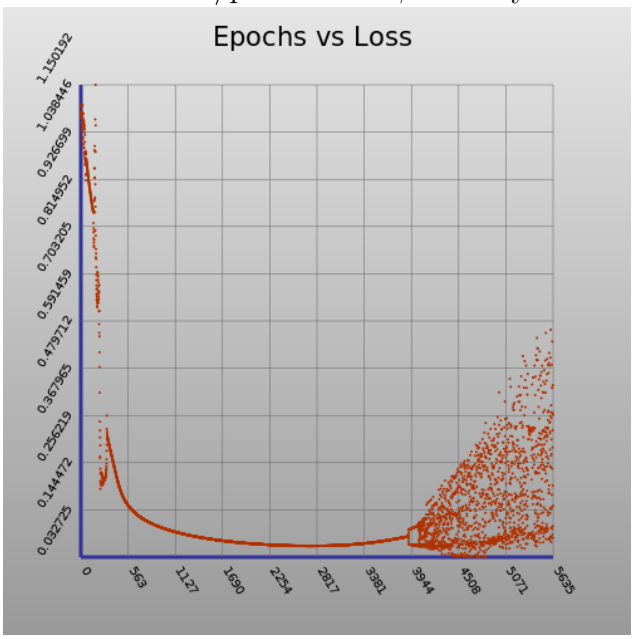


Figure 4: Gráfica costo/pérdida con $\beta = 0.9$



Finalmente, se contrasta el tamaño de batch, realizando un entrenamiento con $\beta = 0.9$ pero ahora promediando el gradiente cada 5 ejemplos de entrenamiento, es decir tamaño batch de 5.

Figure 5: Gráfica costo/pérdida con $\beta = 0.9$ y $batch\_size = 5$



0.4 Conclusión

En cuanto a los resultados obtenidos, se pudo observar que tanto el término de momento como el promediar los gradientes influyeron bastante a la hora de realizar el entrenamiento, ya que dieron mayor estabilidad en el descenso de la función de costo. De manera más específica, mover el término de momento hizo que la convergencia se diera de manera más rápida, mientras que el promediar los gradientes en un mayor número de ejemplos hizo que el valor de la función de costo no variara demasiado.

También se hace más evidente la ventaja de tener una buena organización de código, ya que, las variantes de gradiente descendiente implementadas, son sólo algunas de las muchas que se pueden implementar y que son mucho más complejas, por lo que se hace más necesario abstraer los detalles de la implementación para poder enfocarse en el problema que se quiere resolver.

Algunas mejoras que se podrían implementar, es usar una taza de aprendizaje o learning rate adaptativo, ya sea con técnicas sencillas done se usa una función que depende sólo del número de iteraciones como exponential decay o con alguna técnica más compleja como adagrad o adam.

Finalmente, también se podría considerar algún método de regularización para tratar de prevenir o reducir el sobreajuste a los datos, usando técnicas como dropout o early stopping.