



Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Estado de México

Materia:

Inteligencia artificial avanzada para la ciencia de datos I (Gpo 101)

Implementación de una técnica
de aprendizaje máquina sin el uso
de un framework.

Santiago Espinosa Domínguez

A01747478

Jorge Adolfo Ramírez Uresti

1 de septiembre del 2024

Para implementar una técnica de aprendizaje automático, decidí desarrollar una red neuronal que incluye los procesos de *forward propagation* y *backward propagation*, utilizando la función de activación sigmoide. La red divide el conjunto de datos en entrenamiento, validación y prueba. Durante el proceso de entrenamiento, la red se entrena utilizando los datos de entrenamiento y validación, y al final, se evalúa su exactitud con los datos de prueba.

Los datos fueron obtenidos del siguiente enlace en Kaggle:

<https://www.kaggle.com/code/irfanyakut/seeds-veri-seti-zerinde-pca-analizi-with-python>.

Este conjunto de datos describe las características de las semillas de trigo, las cuales se clasifican en tres tipos distintos según sus atributos.

Lo primero que se realizó fue la importación de las librerías necesarias, utilizamos las librerías de pandas, numpy y matplotlib.



```
1 #Importamos las librerías necesarias
2 import pandas as pd
3 import numpy as np
4 import matplotlib.pyplot as plt
```

Importación de librerías

Luego cargamos los datos de las semillas en un DataFrame y luego le agregamos etiquetas a las columnas.

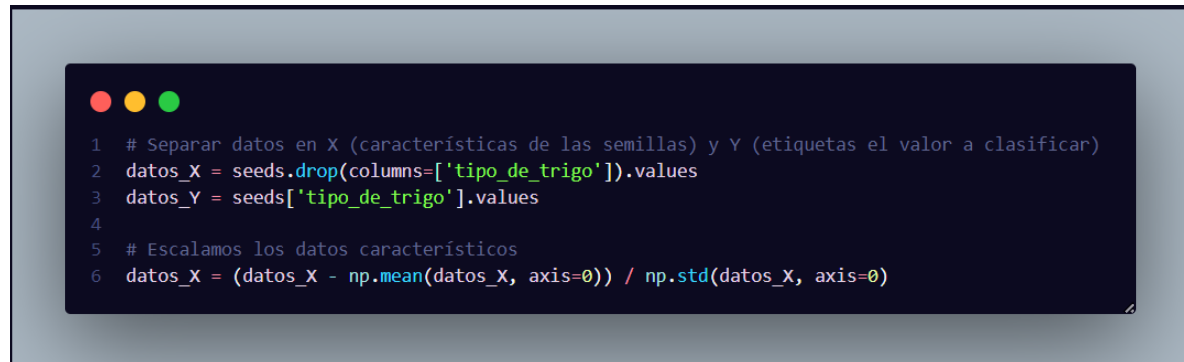


```
1 # Cargamos los datos de un archivo de texto proporcionado por kaggle
2 seeds = pd.read_csv("seeds_dataset.txt", header=None, sep="\t")
3 # Le ponemos etiquetas a las columnas
4 seeds.columns = ["area", "ambiente", "compacidad", "longitud_del_nucleo",
5                  "ancho_nucleo", "coeficiente_asimetrico", "longitud_del_agujero_del_nucleo",
6                  "tipo_de_trigo"]
```

Configuración de los datos

Después de tener los datos, tenemos que dividirlos entre las características que usaremos para entrenar el modelo y la variable que se desea clasificar. Para esto guardaremos las características de las semillas en la variable “datos_X” esto lo logramos copiando todo el Dataset a excepción de la columna “tipo de trigo” ya que esta es la variable por clasificar, y para las etiquetas se agrega solo la columna “tipo de trigo”. Luego escalamos los datos de

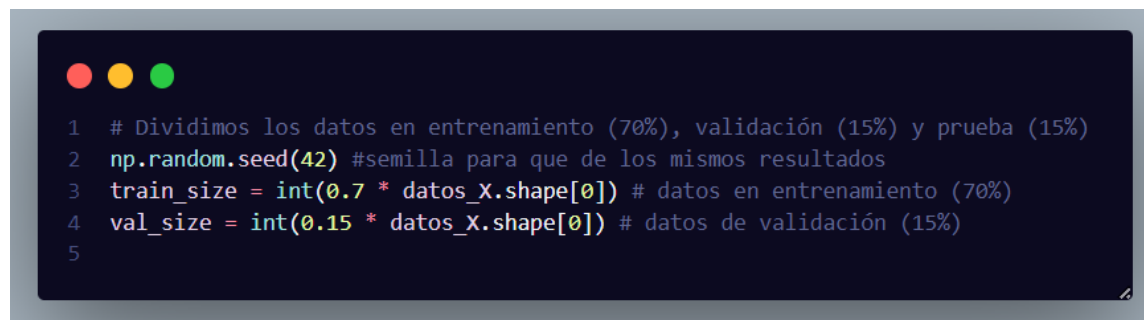
las características para que estén dentro de un rango similar para mejorar la convergencia de la red al momento de entrenarla.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains six lines of Python code for data preprocessing.

```
1 # Separar datos en X (características de las semillas) y Y (etiquetas el valor a clasificar)
2 datos_X = seeds.drop(columns=['tipo_de_trigo']).values
3 datos_Y = seeds['tipo_de_trigo'].values
4
5 # Escalamos los datos característicos
6 datos_X = (datos_X - np.mean(datos_X, axis=0)) / np.std(datos_X, axis=0)
```

Separación y escalamiento de los datos

Una vez con los datos preparados para ser procesados, dividimos los datos en entrenamiento, validación y prueba, se decidió un 70% de entrenamiento ya que normalmente se asigna una gran parte de los datos al entrenamiento porque el modelo necesita suficiente información para aprender patrones representativos del conjunto de datos. Se escogió tener un grupo de datos del 15% de los datos para ajustar hiperparámetros y evaluar el modelo durante el proceso de entrenamiento. Y finalmente el ultimo 15% de los datos es para el conjunto de prueba ya que es necesario que sea lo suficientemente grande como para proporcionar una buena evaluación del rendimiento del modelo en datos nuevos.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains five lines of Python code for splitting the data.

```
1 # Dividimos los datos en entrenamiento (70%), validación (15%) y prueba (15%)
2 np.random.seed(42) #semilla para que de los mismos resultados
3 train_size = int(0.7 * datos_X.shape[0]) # datos en entrenamiento (70%)
4 val_size = int(0.15 * datos_X.shape[0]) # datos de validación (15%)
5
```

Separación de los datos

Una vez obtenidos los tamaños de los datos de entrenamiento, validación y prueba se randomizan los índices para que todos los conjuntos de datos tengan los 3 tipos de trigo variados, ya aleatorizados los índices se guardan en variables los índices de cada grupo de datos y luego se obtienen los datos tanto de las características del trigo como del trigo de cada conjunto de datos.

```

1 # Randomiza los indices para los datos de entrenamiento, validación y prueba
2 indices = np.random.permutation(datos_X.shape[0])
3 train_indices = indices[:train_size] # indices de entrenamiento
4 val_indices = indices[train_size:train_size+val_size] # indices de validación
5 test_indices = indices[train_size+val_size:] # indices de prueba
6
7 # Se obtienen los datos en los indices y se separan
8 X_train, Y_train = datos_X[train_indices], datos_Y[train_indices]
9 X_val, Y_val = datos_X[val_indices], datos_Y[val_indices]
10 X_test, Y_test = datos_X[test_indices], datos_Y[test_indices]
11

```

Obtención de los datos de entrenamiento, validación y prueba

Una vez con los datos han sido obtenidos, normalizados y separados se crea una clase de red neuronal, la cual va a tener varias funciones útiles para poder entrenar. Esta clase requiere de el tamaño de los datos de entrada, el tamaño de la capa oculta y el tamaño de la salida, se creo de este modo para que pueda utilizarse con varios Data sets. Algo importante de la red neuronal es definir los pesos y los sesgos entre las capas, para esto se usa la función de numpy “random” para generar los pesos aleatorios, el tamaño del arreglo va a depender del tamaño de los datos de entrada y de la capa oculta, para el sesgo es un arreglo de ceros que van solo en las neuronas que están en la capa oculta. Para los pesos y sesgos entre la capa oculta y la capa de salida funciona exactamente igual, solo que con sus respectivos valores.

```

1 # Se crea una clase de red neuronal
2 class NeuralNetwork:
3     def __init__(self, input_size, hidden_size, output_size):
4         # Inicialización de pesos y sesgos
5
6         #Pesos entre la capa de entrada y la capa oculta
7         self.W1 = np.random.randn(input_size, hidden_size)
8         self.b1 = np.zeros((1, hidden_size))
9         #Pesos entre la capa oculta y la capa de salida
10        self.W2 = np.random.randn(hidden_size, output_size)
11        self.b2 = np.zeros((1, output_size))

```

Creación de la red neuronal y definición de pesos iniciales

Las primeras funciones que se definieron fueron la clasificación de *sigmoid* que convierte el valor de entrada en un valor que varía entre 0 y 1, y también se define su derivada que se va a utilizar en *backward propagation*.

```
1 def sigmoid(self, z):
2     # Función de activación Sigmoid. Convierte la entrada en un valor entre 0 y 1.
3     return 1 / (1 + np.exp(-z))
4
5 def sigmoid_derivative(self, z):
6     # Derivada de la función Sigmoid, usada en backpropagation
7     return z * (1 - z)
```

Definición de funciones *sigmoid* y *sigmoid_derivative*

Una de las funciones más importantes es *forward propagation*, ya que su función es calcular la salida de la capa de salida, es decir, calcular el resultado. Lo que requiere esta función es algún conjunto de datos y su proceso es multiplicar los datos por los pesos y suma los sesgos de la capa oculta, luego el resultado pasa a la función *sigmoid* una vez pasa esto se repite el proceso, pero ahora con el resultado obtenido por la función *sigmoid*. Al final se retorna el último resultado siendo este la salida final de la red.

```
1     def forward(self, X):
2         # Forward Propagation
3         # Calcula la salida de la capa oculta
4         self.z1 = np.dot(X, self.W1) + self.b1
5         self.a1 = self.sigmoid(self.z1)
6         # Calcula la salida de la capa de salida
7         self.z2 = np.dot(self.a1, self.W2) + self.b2
8         self.a2 = self.sigmoid(self.z2)
9         return self.a2 # Retorna la salida final de la red
10
```

Función *forward*

La función de *backward propagation* es crucial en el aprendizaje automático, ya que su objetivo principal es calcular el error entre la salida real y la predicción de la red neuronal,

para ajustar los pesos y sesgos en las capas ocultas y de salida. Este proceso permite que la red aprenda y mejore su precisión con el tiempo.

Los parámetros que requiere esta función incluyen las características de entrada, las etiquetas de los tipos de trigo, la salida generada por el *forward propagation*, y la tasa de aprendizaje. El primer paso de la función es calcular el error restando la salida de la red y las etiquetas reales. A continuación, se obtiene el gradiente de la capa de salida multiplicando este error por la derivada de la función sigmoide aplicada a la salida. Luego, se calculan las derivadas de la función de pérdida con respecto a los pesos y sesgos de la capa de salida.

El gradiente de la capa oculta se obtiene multiplicando el gradiente de la capa de salida por los pesos de la capa de salida y la derivada de la función sigmoide aplicada a la capa oculta. Este resultado muestra cómo el error en la capa de salida afecta a la capa oculta. Posteriormente, la derivada de la pérdida con respecto a los pesos de la capa oculta se calcula multiplicando la transpuesta de los datos de entrada por el gradiente del error en la capa oculta. Finalmente, los pesos y sesgos se actualizan utilizando el gradiente calculado y la tasa de aprendizaje, permitiendo que la red neuronal ajuste sus parámetros para minimizar el error en futuras predicciones.

```
1 def backward(self, X, Y, output, learning_rate):
2     # Backward Propagation
3     # Calcula el error entre la salida real y la predicción
4     error = output - Y
5     # Gradiente de la capa de salida
6     d_z2 = error * self.sigmoid_derivative(output)
7     # Derivada de los pesos y sesgos de la capa de salida
8     d_w2 = np.dot(self.a1.T, d_z2)
9     d_b2 = np.sum(d_z2, axis=0, keepdims=True)
10
11    # Gradiente de la capa oculta
12    d_z1 = np.dot(d_z2, self.W2.T) * self.sigmoid_derivative(self.a1)
13    # Derivada de los pesos y sesgos de la capa oculta
14    d_w1 = np.dot(X.T, d_z1)
15    d_b1 = np.sum(d_z1, axis=0, keepdims=True)
16
17    # Actualización de los pesos y sesgos utilizando el gradiente y la tasa de aprendizaje
18    self.W1 -= learning_rate * d_w1
19    self.b1 -= learning_rate * d_b1
20    self.W2 -= learning_rate * d_w2
21    self.b2 -= learning_rate * d_b2
```

Función *backward*

La función de entrenamiento es crucial en este proceso de aprendizaje ya que es donde se van a activar las funciones de *forward* y *backward*. Los parámetros que requiere la función son los datos de entrenamiento, y los de validación tanto las características como las

etiquetas de los tipos de trigo, de igual manera requiere el número de épocas y la tasa de aprendizaje.

El proceso es muy sencillo ya que es solo un ciclo “for” que va iterar tantas veces como sean las épocas decididas para el entrenamiento, entonces entra en un ciclo llamando a la función *forward* para obtener la salida y luego mandarla a la función *backward* para que se actualicen los pesos. Para ir checando el proceso de aprendizaje, cada 100 épocas se calcula el error tanto en los datos de entrenamiento con los datos de validación sacando el promedio del cuadrado de la diferencia de las etiquetas y la salida.

```
1 def train(self, X_train, Y_train, X_val, Y_val, epochs, learning_rate):
2     # Entrenamiento de la red neuronal a lo largo de varias épocas
3     for epoch in range(epochs):
4         # Paso de forward propagation
5         output = self.forward(X_train)
6         # Paso de backward propagation y actualización de parámetros
7         self.backward(X_train, Y_train, output, learning_rate)
8
9         # Cada 100 épocas, calcula y muestra las pérdidas (losses) de entrenamiento y validación
10        if epoch % 100 == 0:
11            train_loss = np.mean(np.square(Y_train - output))
12            val_output = self.forward(X_val)
13            val_loss = np.mean(np.square(Y_val - val_output))
14            print(f'Epoch {epoch}, Training Loss: {train_loss}, Validation Loss: {val_loss}')
15
```

Función *train*

La ultima función de la red neuronal es la de *predict* en ella lo único que hace es retornar la salida de la función *forward* de un conjunto de datos, esta salida es redondeada para obtener las clases discretas.

```
1 def predict(self, X):
2     # Realiza predicciones sobre nuevos datos, usando la red entrenada
3     output = self.forward(X)
4     # Redondea las salidas para obtener clases discretas
5     return np.round(output)
```

Función *predict*

Para realizar las predicciones, las etiquetas se convierten a *one-hot encoding* para transformar las etiquetas categóricas en vectores binarios. Por ejemplo, si una muestra pertenece a la clase 2, el vector resultante será [0, 1, 0], donde el 1 indica la clase a la que pertenece la muestra.

Con las etiquetas en *one-hot encoding*, podemos inicializar y entrenar la red neuronal. El tamaño de entrada será igual al número de características en los datos de entrenamiento. Para la capa oculta, hemos elegido 10 neuronas, aunque este número puede variar según la complejidad deseada para el modelo. Finalmente, el tamaño de salida será de 3, correspondiente a los tres tipos de trigo que la red debe clasificar. La función de activación sigmoide se utiliza en las capas de la red para clasificar las salidas en 0 o 1, lo que facilita la interpretación de los resultados como probabilidades para cada clase.

```
1 # Convertir etiquetas en one-hot encoding
2 Y_train_one_hot = np.eye(3)[Y_train - 1]
3 Y_val_one_hot = np.eye(3)[Y_val - 1]
4 Y_test_one_hot = np.eye(3)[Y_test - 1]
5
6 # Inicializar y entrenar la red neuronal
7 input_size = X_train.shape[1] # características
8 hidden_size = 10 # Número de neuronas en la capa oculta
9 output_size = 3 # Tres tipos de trigo
```

Preparación de los datos para la red neuronal

Una vez todos los datos están preparados se crea una red neuronal con los tamaños de las neuronas y se entrena con los datos de entrenamiento y validación, se entrena con 1000 épocas y con una tasa de aprendizaje 0.01. Una vez entrenada la red se obtiene la predicción con los datos de prueba y se obtiene la exactitud del modelo.

```
1 nn = NeuralNetwork(input_size, hidden_size, output_size) # se crea la Red neuronal
2 # se entrena con los valores de entrenamiento y validación
3 nn.train(X_train, Y_train_one_hot, X_val, Y_val_one_hot, epochs=1000, learning_rate=0.01)
4 # Evaluar con el conjunto de prueba
5 predictions = nn.predict(X_test)
6 accuracy = np.mean(np.argmax(predictions, axis=1) == (Y_test - 1))
7 print(f'Test Accuracy: {accuracy * 100}%')
```

Creación de la red neuronal, entrenamiento de la red y su predicción.

Para checar el rendimiento de la red neuronal se decidió ocupar la matriz de confusión, ya que muestra cuántas veces cada clase verdadera fue clasificada correcta o incorrectamente en otra clase.

Entonces lo primero que se hace es convertir las predicciones de la red neuronal desde su formato *one-hot encoding* de vuelta a etiquetas de clases. Luego, se construye una matriz de confusión que compara las etiquetas verdaderas con las etiquetas predichas.

Para crear la matriz de confusión primero se debe de inicializar una matriz de ceros, cuyo tamaño depende del número de clases en el conjunto de datos. A continuación, mediante un ciclo “for”, se recorre cada par de etiqueta verdadera y predicha, incrementando el valor correspondiente en la matriz de confusión.

Finalmente, el código visualiza la matriz de confusión utilizando matplotlib. La matriz se representa con una escala de colores donde cada celda muestra el número de predicciones realizadas para esa combinación de etiqueta verdadera y predicha. Además, se agregan etiquetas a los ejes y se imprimen los valores en cada celda para facilitar la interpretación de los resultados.

```
1 # Convertir de one-hot encoding a etiquetas
2 predictions_labels = np.argmax(predictions, axis=1) + 1
3
4 # Calcular la matriz de confusión
5 num_classes = len(np.unique(Y_test))
6 confusion_matrix = np.zeros((num_classes, num_classes), dtype=int)
7
8 for true_label, predicted_label in zip(Y_test, predictions_labels):
9     confusion_matrix[true_label-1, predicted_label-1] += 1
10
11 # Visualizar la matriz de confusión
12 plt.figure(figsize=(8, 6))
13 plt.imshow(confusion_matrix, cmap="Blues")
14 plt.title("Confusion Matrix")
15 plt.colorbar()
16 plt.xlabel("Predicted Labels")
17 plt.ylabel("True Labels")
18
19 # Añadir las etiquetas de los ejes
20 classes = np.arange(1, num_classes+1)
21 plt.xticks(classes - 1, classes)
22 plt.yticks(classes - 1, classes)
23
24 # Añadir valores a cada celda
25 for i in range(num_classes):
26     for j in range(num_classes):
27         plt.text(j, i, confusion_matrix[i, j], ha="center", va="center", color="black")
28
29 plt.show()
```

Matriz de confusión

A continuación, se presentaran los resultados de la pérdida de la fase de entrenamiento y validación en cada 100 épocas, esto para que no se llene la terminal de muchos datos.

```

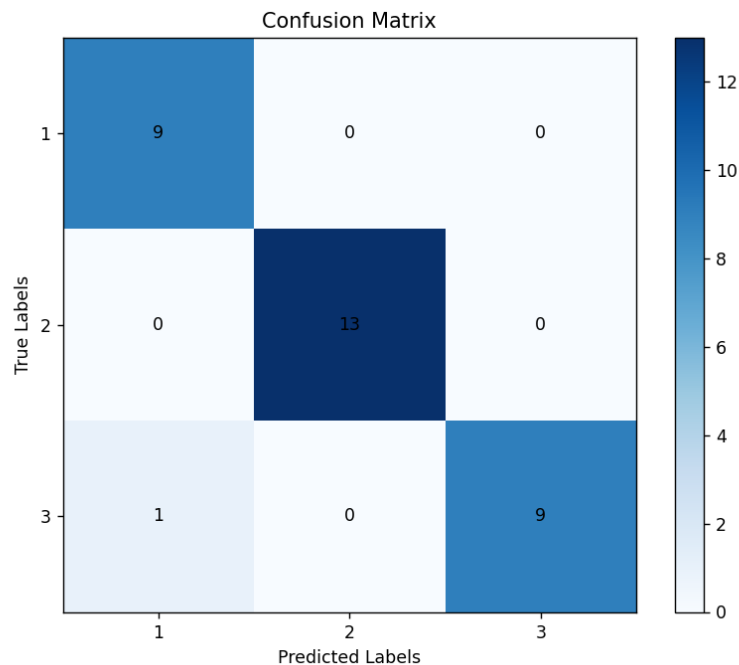
Epoch 0, Training Loss: 0.3170315392900614, Validation Loss: 0.28607227250954936
Epoch 100, Training Loss: 0.056860777956867066, Validation Loss: 0.046017295374290744
Epoch 200, Training Loss: 0.04453616790562551, Validation Loss: 0.03442934318272248
Epoch 300, Training Loss: 0.03814568950002661, Validation Loss: 0.030956504752587585
Epoch 400, Training Loss: 0.033755910611602574, Validation Loss: 0.02902739427005055
Epoch 500, Training Loss: 0.030361314703533065, Validation Loss: 0.02751487966979615
Epoch 600, Training Loss: 0.027479242231245177, Validation Loss: 0.026304280874820334
Epoch 700, Training Loss: 0.02506698795248213, Validation Loss: 0.025338576661146056
Epoch 800, Training Loss: 0.023051056511453172, Validation Loss: 0.024511324820935244
Epoch 900, Training Loss: 0.021307067047211692, Validation Loss: 0.023762351834429436
Test Accuracy: 96.875%

```

Resultado de los datos de entrenamiento, validación y prueba

Como se puede observar, la pérdida tanto en el conjunto de entrenamiento como en el de validación disminuye progresivamente a lo largo de las épocas, alcanzando valores muy bajos. Esto indica que el modelo ha logrado aprender de manera efectiva las características distintivas de las semillas de trigo. La alta precisión en la fase de prueba, del 96.875%, refuerza la idea de que el modelo generaliza bien, es decir, que su rendimiento en datos no vistos es excelente.

Sin embargo, también se observa que la reducción en la pérdida se vuelve marginal después de aproximadamente 600 épocas. Este comportamiento sugiere que el modelo ha alcanzado su capacidad máxima de aprendizaje, y que continuar entrenando con el mismo conjunto de datos probablemente no resultará en mejoras significativas. Esto podría ser una señal de que el modelo está cerca de sobreajustarse a los datos de entrenamiento.



Resultados de la matriz de confusión

La matriz de confusión muestra un alto rendimiento en la clasificación de las semillas de trigo, logrando clasificar correctamente la mayoría de las muestras en sus respectivas clases. Sin embargo, se observa un caso de error donde una semilla de trigo perteneciente a la clase 3 fue clasificada incorrectamente como perteneciente a la clase 1. Este tipo de error podría sugerir similitudes en las características de las semillas de las clases 1 y 3, lo que podría ser un área de interés para un análisis más detallado. En general, el modelo demuestra una capacidad de clasificación robusta, aunque con espacio para una pequeña mejora en la precisión para evitar estos errores aislados.