MLP.

Multi-Layer Perceptron.

Clase.

Inteligencia Artificial II

Sección.

D01

Profesora.

Arana Daniel Nancy Guadalupe.

Equipo.

Grover Morales Francisco Javier.

Gutiérrez Alatorre André Ángel Humberto.

Robles Perez Cristian Ismael.

22 de marzo de 2022

Algoritmo.

Descripción.

El perceptrón multicapa es una clase de red neuronal formada, como su nombre lo indica, por varias capas de neuronas (perceptrones o adaline) interconectadas que tiene la capacidad para resolver problemas que no son linealmente separables.

En el año de 1969, Minsky y Papert demostraron que el perceptrón simple o el Adaline no pueden resolver esa cierta clase de problemas. Para esto, en el Perceptrón Multicapa, a través de la Regla Delta Generalizada retro-propagando el error a las capas anteriores pueden ajustarse los pesos de todas las neuronas y se puede llegar a ser el aproximador universal de problemas que no son linealmente separables con resultados bastante buenos.

Por el otro lado, está clase de red neuronal cae en la limitación que si no se es propiamente entrenada, esto también aunado a que la existencia de mínimos locales llegan a dificultar llegamos a salidas altamente imprecisas.

Código.

Función de entrenamiento

```
def train(self, X: list, Y: list, max_epoch: int, min_error: float, batch: bool):
    """Se entrena el mlp usando feed_forward y backpropagation"""
    # Se calcula la cantidad de filas m
    m = len(X)
    # vector de datos correctos y codificado
    D = self.encode_desired_output(Y)

    # Error cuadrático medio (mse)
    mean_sqr_error = 0
    # Lista de errores cuadráticos medios
    mse_list = []
    # Error cuadrático acumulado por época
    epoch_sqr_error = 0
    # Número de épocas
    epoch = 0
    # acumulador de sensibilidades
    s = [0 for i in range(len(self.sensitivities))]
```

```
        while True:
            # Se itera por cada fila de X
            for i in range(m):
                y = self.feed_forward(X[i])
                error = np.subtract(D[i], y)
                epoch_sqr_error += np.sum(error ** 2)

                # Se calculan las sensibilidades
                self.get_sensitivity(error)

                # Se ajustan los pesos
                if not batch:
                    self.backpropagation(X[i])
                else:
                    s = np.add(np.array(self.sensitivities), s)

            # Se imprimen los pesos de la capa oculta por época
            if self.plot_weights != None:
                self.plot_weights(self.W_inputs, 'g')

            if batch:
                self.sensitivities = np.divide(s, m)
                for i in range(m):
                    self.backpropagation(X[i])

            # Se obtiene la media del error cuadrático y hacemos que el mse sea cero
            mean_sqr_error = epoch_sqr_error / m
            print(f'Epoca: {epoch} | Error cuadrático: {mean_sqr_error}')
            mse_list.append(mean_sqr_error)
            epoch_sqr_error = 0
            s = [0 for i in range(len(self.sensitivities))]
            epoch += 1

            if self.plot_mse != None:
                self.plot_mse(mse_list)

            # Si se llegó al número máximo de épocas o si el mse es menor al error mínimo
deseado
            if epoch == max_epoch or mean_sqr_error < min_error:
                break

        self.get_confusion_matrix(X, y, D)
        return epoch, mean_sqr_error
```

Funciones de evaluación.

```
def encode_desired_output(self, Y: list):
    """Retorna una matriz de valores codificados para representar los valores
    del vector de valores deseados Y"""
    D = np.zeros((len(Y), len(np.unique(Y))))
    for i in range(len(Y)):
        # TODO: Cambiar esto para que se puedan poner clases no consecutivas
        D[i, Y[i]] = 1
    return D

def encode_guess(self, y):
    """Devuelve el valor más alto de una salida de la red"""
    return np.where(y == np.amax(y))[0][0]
```

Función de transferencia

```
def sigmoid(self, y):
    """Calcula el valor de activación"""
    return 1 / (1 + np.exp(-y))
```

Funciones de barrido

```
def plot_point(self, point: tuple, alpha=None, cluster=None):
    """Toma un array de tuplas y las añade los puntos en la figura con el
    color de su cluster"""
    plt.figure(1)
    cmap = get_cmap('flag')
    if (cluster == None): # clase desconocida
        plt.plot(point[0], point[1], 'o', color='k')
    elif alpha != None: # degradado
        color = cmap(float(int(cluster)/100))
        offset = 0.25
        alpha = (alpha - offset) if (alpha - offset) > 0 else 0
            plt.plot(point[0], point[1], 'o', color=color, markersize=7, markeredgewidth=0,
alpha=alpha)
    else: # entrenamiento
        color = cmap(float(int(cluster)/100))
                plt.plot(point[0], point[1], 'o', markeredgecolor='k', markeredgewidth=1.5,
color=color)


def plot_gradient(self):
    """gráfica el degradado de las clases"""
    x = np.linspace(self.ax_min, self.ax_max, 40)
```

```
y = np.linspace(self.ax_min, self.ax_max, 30)

for i in range(len(x)):
    for j in range(len(y)):
        res = self.mlp.feed_forward((x[i], y[j]))
        cluster = self.mlp.encode_guess(res)
        self.plot_point((x[i], y[j]), res[cluster], cluster)
```

Análisis comparativo.

Igualmente que las dos implementaciones anteriores, esta red neuronal se trata de un algoritmo de clasificación supervisado, recibiendo un patrón de entrada con las clasificaciones correctas para finalmente, después de ser entrenado poder evaluar nuevos conjuntos de datos que no se les haya dado una clasificación previa.

El adaline llegaba a una aproximación de clasificación gradual y lineal precisa, pero sujeto al funcionamiento lento de una sola neurona. En contraparte, un MLP es la concatenación por capas de «n» neuronas (adaline o perceptrón), donde puede llegar a ser computacionalmente más caro pero podemos llegar a resultados aceptables medianamente rápido; y, dentro del punto más importante de este algoritmo, es capaz de realizar la clasificación de problemas que no son linealmente separables, y, por lo tanto, capaz de resolver problemas donde no solo se tengan dos clases, sino que puedan evaluar muchas más.

Repositorio.

https://github.com/Inteligencia-Artificial-II/MLP