Antonio Gallego Peñalver

Documentación Red Neuronal Multicapa

Clase NetNode

```
class NetNode(object):
    """ Base class that represents a node in a neural net

    def __init__(self):
        self.inputs = []
        self.weights = []
        self.value = None
```

Representa una neurona de una red neuronal. Esta clase define un nodo dentro de la red neuronal. Más abajo está el constructor de la clase, que se ejecuta al crear un nodo.

- inputs guarda los nodos que están conectados a este nodo como entrada.
- weights guarda los pesos asociados a cada conexión de entrada.
- value almacena el valor actual del nodo, el resultado de la función de activación. Se pone None porque no se calculó todavía

Clase Network

```
class Network(object):
    """ Main class to construct ant train neural networks """
```

Representa una red neuronal. Se construyen las capas, se conectan a los nodos, hacen la predicción y el entrenamiento.

Constructor __init__

```
class Network(object):
    """ Main class to construct ant train neural networks """

    def __init__(self, layers):
        """
        Parameters
        ----------
        layers:
            A list that represents the neurons and layers of the network.
            For example, [2, 3, 1] represents a network with 3 layers:
            - input layer: 2 neurons.
            - hidden layer: 3 neurons.
            - output layer: 1 neuron.
        """
        self.net = [[NetNode() for _ in range(size)] for size in layers]

        sizes = len(layers)

        # Make connections
        for layer in range(1, sizes):
            for node in self.net[layer]:
                for unit in self.net[layer - 1]:
                    node.inputs.append(unit)
                    node.weights.append(0)
```

El parámetro layers guarda cada elemento que representa el número de neuronas de cada capa.

- self.net es una lista de listas de nodos (de NetNode). Cada sublista representa una capa de la red. Recorre cada numero size en layers y crea size nodos por capa
- sizes = len(layers) guarda el total de capas de la red

El bucle hace que conecten los nodos de cada capa con los de la anterior.

- layer para cada nodo en la capa actual
- layer – 1 para recorrer todos los nodos de la capa anterior
- inputs para agregar como entrada, asignándole 0 como peso inicial en weights

Función ReLU

```
def relu(self, z):
    """ Relu activation function """
    return max(0, z)
```

Es una función de activación ReLU, devuelve 0 si z es negativo o z si es positivo. Se usa para introducir no linealidad y hace que sea eficiente

Función relu_prime

```
def relu_prime(self, z):
    """ Derivative of relu activation function """
    return 1 if z > 0 else 0
```

Es una función que devuelve la derivada de ReLU. Devuelve 1 si z es mayor que 0 (z > 0) ó 0 si no se da el caso. Se usa durante la retropropagación para ajustar los pesos.

Función predict

```
def predict(self, input_data):
    inputs = self.net[0]

    # Initialize inputs
    for v, n in zip(input_data, inputs):
        n.value = v

    # Forward step
    for layer in self.net[1:]:
        for node in layer:
            in_val = [n.value for n in node.inputs]
            unit_value = np.dot(in_val, node.weights)
            node.value = self.relu(unit_value)

    outputs = self.net[-1]
    return outputs.index(max(outputs, key=lambda node: node.value))
```

Hace una predicción con la red neuronal entrenada. Recibe un valor de entrada y devuelva la clase predicha.

```
inputs = self.net[0]

# Initialize inputs
for v, n in zip(input_data, inputs):
    n.value = v
```

inputs = self.net[0] inicializa los valores de los nodos de entrada en la capa 0, con los datos de entrada de input_data

```
for layer in self.net[1:]:
    for node in layer:
        in_val = [n.value for n in node.inputs]
        unit_value = np.dot(in_val, node.weights)
        node.value = self.relu(unit_value)

outputs = self.net[-1]
return outputs.index(max(outputs, key=lambda node: node.value))
```

Esto es un bucle que permite hacer una propagación hacia delante.

- Se recorre cada valor del vector de entrada (input_data) con el nodo de la capa de entrada. Asigna el valor atributo .value de cada nodo de entrada. Hace que inicialice la red con la entrada actual.
- For layer in self.net comienza la propagación hacia delante. Recorre todas las capas menos la de entrada (capas ocultas y las de salida).
- For node in layer recorre cada nodo de la capa actual para calcular su valor de activación.
- In_val extrae los valores de los nodos de entrada conectados a un nodo. Crea una lista de valores para calcular la activación del nodo.
- Unit_value = np.dot() calcula la suma ponderada de las entradas del nodo.
- Node.value aplica la función de activación ReLU al valor calculado unit_value, y guarda el valor de ese nodo

- Outputs = self.net[-1] se usa para tomar la última capa de la red (capa de salida).
- Retorna el índice del nodo buscado con mayor valor de salida

Función accuracy

```
def accuracy(self, examples):
    correct = 0

    for x_test, y_test in examples:
        prediction = self.predict(x_test)

        if (y_test[prediction] == 1):
            correct += 1

    return correct / len(examples)
```

Esta función calcula la precisión de la red (accuracy). Se utiliza para evaluar que tan buena es la red. Comienza por 0 para usarlo de contador de aciertos

El bucle for x_test, y_test in examples recorre los pares en la lista de examples

- x_test es un array de características normalizadas
- y_test es otro array one-hot con la clase correcta
- prediction llama a la función predict para obtener la clase predicha

Luego se compara con la salida esperada. El if verifica si en el array y_test contiene un 1. Si acierta, suma un uno al contador correct.

Finalmente, devuelve un cálculo del porcentaje de aciertos. Devuelve el 93.3%

Función backpropagation

```
def backpropagation(self, eta, examples, epochs):
    inputs = self.net[0]
    outputs = self.net[-1]
    layer_size = len(self.net)
```

Se usa para hacer el entrenamiento de la red con la retropropagación

- eta es la tasa de aprendizaje
- examples es la lista de pares
- epochs es el número de veces que se entrena con todos los ejemplos
- inputs apunta a la capa de entrada
- outputs a la capa de salida
- layer_size es el número total de capas

```
for layer in self.net[1:]:
    for node in layer:
        node.weights = [np.random.uniform()
                        for _ in range(len(node.weights))]
```

Este bucle inicializa los pesos. Se inicializan los pesos con valores aleatorios entre 0 y 1.

```
        for epoch in range(epochs):
            for x_train, y_train in examples:
                # Initialize inputs
                for value, node in zip(x_train, inputs):
                    node.value = value

                # Forward step
                for layer in self.net[1:]:
                    for node in layer:
                        in_val = [n.value for n in node.inputs]
                        unit_value = np.dot(in_val, node.weights)
                        node.value = self.relu(unit_value)

                # Initialize delta
                delta = [[] for _ in range(layer_size)]

                # Error for the MSE cost function
                err = [y_train[i] -
                       outputs[i].value for i in range(len(outputs))]

                delta[-1] = [self.relu_prime(outputs[i].value) * err[i]
                             for i in range(len(outputs))]

                # Backward step
                hidden_layers = layer_size - 2
                for i in range(hidden_layers, 0, -1):
                    layer = self.net[i]
                    n_layers = len(layer)

                    # Weights from the last layer
                    w = [[node.weights[l] for node in self.net[i + 1]]
                         for l in range(n_layers)]

                    delta[i] = [self.relu_prime(
                        layer[j].value) * np.dot(w[j], delta[i + 1]) for j in
range(n_layers)]
                # Update weights
                for i in range(1, layer_size):
                    layer = self.net[i]
                    in_val = [node.value for node in self.net[i - 1]]
                    n_layers = len(self.net[i])
                    for j in range(n_layers):
                        layer[j].weights = np.add(
                            layer[j].weights, np.multiply(eta * delta[i][j], in_val))
```

Esto es un bucle de entrenamiento por épocas. Se repite el entrenamiento durante el número de épocas asignado.

El bucle for x_train, y_train in examples es un entrenamiento por cada ejemplo. Recorre cada ejemplo de forma individual

- x_train es un array con las características de entrada, normalizadas
- y_train es otro array one-hot con la clase esperada

```
for value, node in zip(x_train, inputs):
    node.value = value
```

Aquí se inicializan los valores de entrada a cada neurona de la capa de entrada

```
for layer in self.net[1:]:
    for node in layer:
        in_val = [n.value for n in node.inputs]
        unit_value = np.dot(in_val, node.weights)
        node.value = self.relu(unit_value)
```

Este bucle hace la propagación hacia delante. Se obtienen los valores de los nodos de entrada, calcula la suma ponderada con los pesos, aplica la función ReLU y el resultado se guarda como valor del nodo

```
delta = [[] for _ in range(layer_size)]
```

Esto es un array de deltas. Almacena los deltas (los errores parciales) para la retropropagación, iniciándose al comienzo en una lista vacía

```
err = [y_train[i] -
       outputs[i].value for i in range(len(outputs))]

delta[-1] = [self.relu_prime(outputs[i].value) * err[i]
             for i in range(len(outputs))]

# Backward step
hidden_layers = layer_size - 2
```

La variable err calcula el error de cada nodo de salida.

delta[-1] calcula los deltas de la capa de salida. Su salida indica cuánto se debe corregir cada nodo de salida

```
                hidden_layers = layer_size - 2
                for i in range(hidden_layers, 0, -1):
                    layer = self.net[i]
                    n_layers = len(layer)

                    # Weights from the last layer
                    w = [[node.weights[l] for node in self.net[i + 1]]
                         for l in range(n_layers)]

                    delta[i] = [self.relu_prime(
                        layer[j].value) * np.dot(w[j], delta[i + 1]) for j in
range(n_layers)]
```

Esto es la retropropagación del error

- hidden_layers = layer_size – 2 son el número de capas ocultas

Se recorren inversamente. Por cada nodo, se calcula el error como una combinación de los deltas de la capa siguiente, y de los pesos que conectan con ella. Se multiplica por la derivada de ReLU al valor del nodo actual.

```
for i in range(1, layer_size):
    layer = self.net[i]
    in_val = [node.value for node in self.net[i - 1]]
    n_layers = len(self.net[i])
    for j in range(n_layers):
        layer[j].weights = np.add(
            layer[j].weights, np.multiply(eta * delta[i][j], in_val))
```

Se actualizan los pesos de cada nodo en las capas ocultas y las de salida. Cada peso se ajusta en cuanto a eta, el delta y los valores de entrada.

Utiliza np.add y np.multiply para calcular el nuevo peso. Que seria el peso actual + tasa de aprendizaje * delta * el valor de entrada.

Al final se muestra el error medio por época

Listar los 10 elementos con los datos importados y su salida

```
# import data to play with
iris_X, iris_y =
datasets.load_iris(return_X_y=True)
# First 10 elements of input data
iris_X[:10]
```

Guardamos las características de Iris en la matriz iris_X. Seleccionamos las primeras 10 lineas de la matriz en el siguiente orden: Longitud de sépalo, ancho del sépalo, longitud del pétalo y ancho del pétalo:

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5,
0.1]])
```

```
# First 10 elements of output
iris_y[:10]
```

iris_y es un array que contiene las etiquetas de clase de cada flor del dataset. Se seleccionan los 10 primeros elementos del array. Siendo 0 Setosa, 1 Versicolor y 2 Virginica (Solo se ven Setosa)

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0,
0])
```

Normalización

```
iris_x_normalized = normalize(iris_X, axis=0)
```

Normaliza los datos de entrada columna por columna (longitud de pétalo, sépalo etc). axis=0 hace que se normalice cada columna independientemente.

Entrenamiento y prueba

```
# Creating train and test data
'''
80% -- train data
20% -- test data
'''
X_train, X_test, y_train, y_test = train_test_split(
    iris_x_normalized, iris_y, test_size=0.2,
shuffle=True)
```

Se divide dos conjuntos en entrenamiento y prueba. shuffle hace que se mezcle aleatoriamente las muestras antes de dividirlas.

- X_train son las entradas de entrenamiento
- X_test son las entradas de prueba
- y_train son las etiquetas de entrenamiento
- y_test son las etiquetas de prueba

Conversión de etiquetas a one-hot

```
# Convert classes from categorical ('Setosa', 'Versicolor', 'Virginica')
# to numerical (0, 1, 2) and then to one-hot encoded ([1, 0, 0], [0, 1, 0], [0, 0,
1]).
[0]--->[1 0 0]
[1]--->[0 1 0]
[2]--->[0 0 1]
'''
# y_train = np_utils.to_categorical(y_train, num_classes=3)
# y_test = np_utils.to_categorical(y_test, num_classes=3)
y_train = to_categorical(y_train, num_classes=3)
y_test = to_categorical(y_test, num_classes=3)
```

Convierte las etiquetas numéricas 0, 1, 2 en arrays one-hot. Es necesario porque la salida de la red tiene 3 neuronas.

Antonio Gallego Peñalver

Creación de red neuronal

```
examples = []
for i in range(len(X_train)):
    examples.append([X_train[i], y_train[i]])


net = Network([4, 7, 3])
net.backpropagation(0.1, examples, 500)
```

Se crea una lista llamada examples, donde cada elemento es una pareja (la entrada normalizada con la one-hot). Se utilizará para entrenar la red.

El segundo bloque de código crea una red neuronal con 4 entradas, 1 capa oculta compuesta por 7 neuronas y 3 neuronas de salida.

Con net.backpropagation, la tasa de aprendizaje será de 0.1 durante 500 épocas, usando los ejemplos de antes
Salida:

```
epoch 0/500 | total error=-1.3776353936327996e-
epoch 1/500 | total error=0.0002710094599961225
epoch 2/500 | total error=0.00031310830730766026
epoch 3/500 | total error=0.000326802834076031
epoch 4/500 | total error=0.0003325084440425591
epoch 5/500 | total error=0.0003372403320414413
epoch 6/500 | total error=0.0003459775953287253
epoch 7/500 | total error=0.0003616113355905615
epoch 8/500 | total error=0.0004108707562767705 7
epoch 9/500 | total error=0.0005468574644217452
epoch 10/500 | total error=0.0007661732200914864
epoch 11/500 | total error=0.0010047928631536284
epoch 12/500 | total error=0.0012426315819774603
epoch 13/500 | total error=0.0014571597504197972
epoch 14/500 | total error=0.001659245369489921
epoch 15/500 | total error=0.0018321234273578485
epoch 16/500 | total error=0.002005754876385904
epoch 17/500 | total error=0.0021653910025266243
epoch 18/500 | total error=0.0023074994371538146
epoch 19/500 | total error=0.0024354827585918422
epoch 20/500 | total error=0.0025461338925799808
epoch 21/500 | total error=0.0026837741902162116
epoch 22/500 | total error=0.0027800064813425646
epoch 23/500 | total error=0.0028995010155745036
epoch 24/500 | total error=0.00300002061542929
...
epoch 496/500 | total
epoch 497/500 | total error=0.0007340[illegible]=0.000734641953298502
epoch 498/500 | total
epoch 499/500 | total
error=0.0007100560516921438
```

Muestra el error medio por cada época

Test de precisión

```
examples = []
for i in range(len(X_test)):
    examples.append([X_test[i], y_test[i]])
```

Se crea un array examples con los datos de prueba

```
accuracy = net.accuracy(examples)
print(f"Accuracy: {accuracy}")
```

Calcula el porcentaje de aciertos de la red con los datos de prueba

Salida (96,6% de precisión):

```
Accuracy: 0.9666666666666667
```

```
prediction = net.predict(X_test[1])
print(f"Desired output: {y_test[1]}")
print(f"Index of output: {prediction}")
```

Se toma una flor del conjunto de prueba para predecirla con su clase en la red entrenada. Muestra la salida esperada con one-hot y el índice que predijo

Salida:

```
Desired output: [0. 0. 1.]
Index of output: 2
```

La flor es Virginica, que también la predijo, pues es la clase 2.