Implementation of Multilayer Back Propagate Neural Network

Project link:
https://colab.research.google.com/drive/1EtsJFt8UQI9GM_fdxBHxg0bjhqieO0X_?usp=sharing

Introduction

This report presents the implementation of multilayer BP neural network, which addresses the multiclassification problems.

BP neural network is a type of supervised machine learning algorithm. It transmits signals forward and errors in reverse. It is one of the most popular supervised machine learning methods that fits both classification and regression tasks. It works like human brain, and in most cases, has significant performance. These become the reasons that I choose this model to address the multiclassification problem.

The implemented BP neural network will take n numerical attributes $X_1, X_2, X_3 \dots X_n$ and will predict the probability $P_{class_1}, P_{class_2}, P_{class_3} \dots P_{class_m}$ of each class that the input data belongs to, then output the index of the class Y that has highest P_{class_Y}.

Multilayer BP Neural Network Description

This section consists of 3 parts: forward propagation of information, back propagation of errors and other optimization techniques including Local Minima and adaptive learning rate.

Forward Propagation

The figure below illustrates the structure of multilayer BP Neural Network, which can be divided into three parts: the input layer, the hidden layer (may have multiple hidden layers), and the output layer.

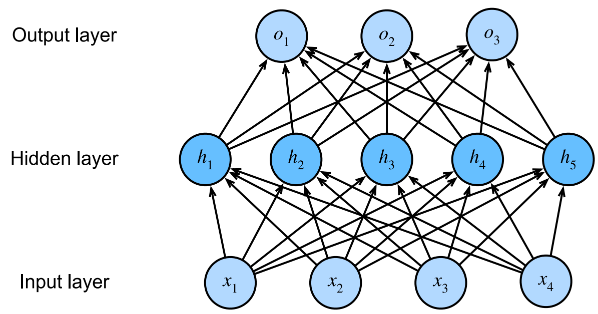

Figure 1 Structure of Mutilayer BP Neural Network (Zhang et al., 2021)

The working process of each node is presented in the following figure. Several input signals are multiplied by their connected weights ***w*** and added to a bias ***b*** , the result is then passed to the activation function ***F(∑ Xi-Wi-b),*** resulting in output ***Y(Y=F (∑ Xi-Wi-b))***. The output Y is then used as input to continue the propagation.

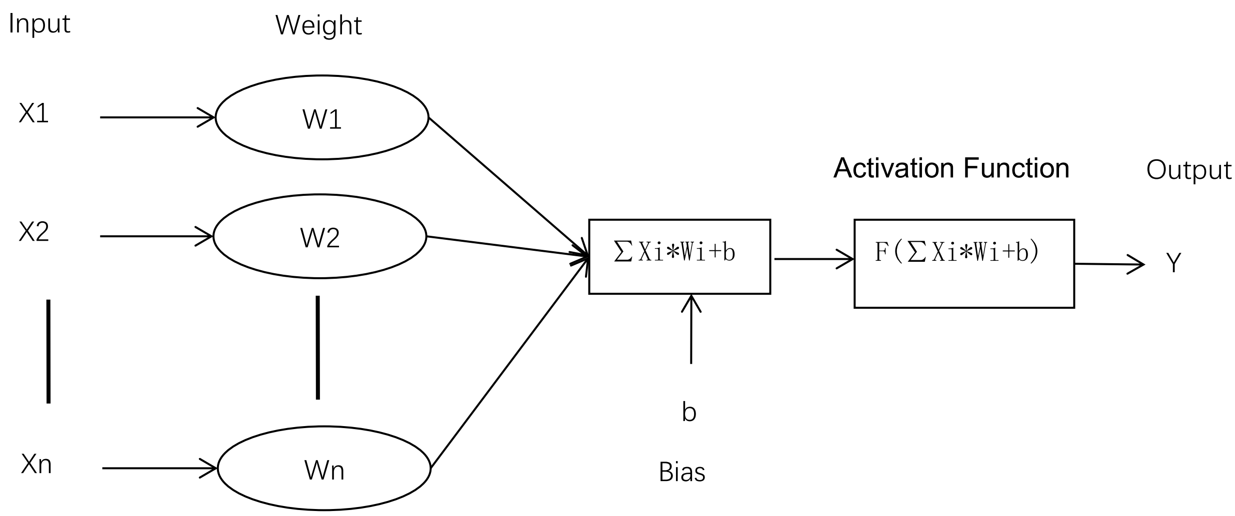

Figure 2 Working process of each node

For the activate function for the hidden layers, there are several available, the two most popular activation functions are **ReLU** and **Sigmoid**. The Equations are listed below:

$$ReLu(x) = \begin{cases} x \ (x > 0) \\ 0 \ (x \leq 0) \end{cases}$$

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

The **ReLU** function has much higher convergence rate as its gradient is either 1 (x>0) or 0 (x<=0). The model with **ReLU** activation function will have much faster training speed. However, the gradient of **ReLU** function will be 0 if the input is lower or equal to 0. **Sigmoid** activation function, on the other hand, is always differentiable but will make the model takes more time to train. The details of the training algorithm will be introduced in next section.

For the activation function for the output layer, the **SoftMax** function is the most popular one for the classification tasks as it can interpret each output $\boldsymbol{x_i}$ into the possibility that the record belongs to class ***i***. Assume we have ***n*** output nodes, the Equation of **SoftMax** function can be listed below:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Back Propagation

After the information is propagated from input layer to hidden layers, and then output layer, the model will produce an output. This output will then be passed to a loss function to measure the error of the model. After that, the error will be used to update the weights and bias in each layer.

The **Cross-entropy** loss function is usually used to measure the errors of **Softmax** outputs. It relies on maximum likelihood estimation and the equation can be listed as following:

$$CrossEntropy(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} y_i \log \hat{y}_i$$

In the equation above, $\mathbf{y}$ is the target values and $\hat{\mathbf{y}}$ is the outputs from **Softmax** function.

The BP Neural Network use gradient descent algorithm to make the error converge to the best state. Given an objective function, the idea is to adjust the input to minimize the output. To do this, we need to calculate the corresponding derivative (gradient) of each input, and then adjust the input toward the negative direction of the gradient.

In the BP network, the objective function is the combination of all the function (activation functions, loss function) that transform the inputs ***x*** into the output ***y***. The gradient descent algorithm adjust the weights and biases to make the objective function has minimal output under the current inputs, that is, $w_{new} = w_{old} + \mu \cdot (-\nabla w_{old})$ and $b_{new} = b_{old} + \mu \cdot (-\nabla b_{old})$, in which μ is learning rate and ∇ is the corresponding gradient.

Assume we have a model that has the following structure:

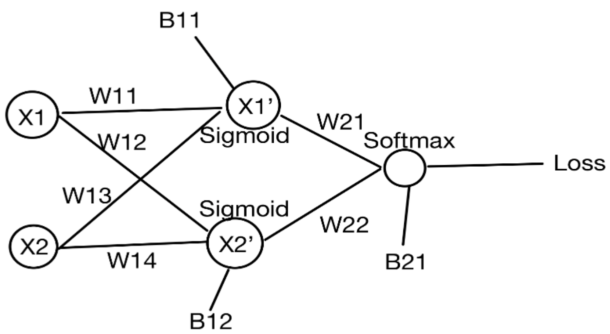

Figure 3 Example Model

The X is input, W is weight, B is bias. The ***outputs of hidden layer*** and ***overall output*** will be as follow:

$$Hidden\ layer\ Output \rightarrow x_1{}' = Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})$$

$$Hidden\ layer\ Output \rightarrow x_2{}' = Sigmoid(w_{12} \cdot x_1 + w_{14} \cdot x_2 + b_{12})$$

$$Output \rightarrow Y = Softmax(w_{21} \cdot x_1{}' + w_{22} \cdot x_2{}' + b_{21})$$

The ***Error*** will be as follow:

$$Error = CrossEntropy(Target, Y)$$

The ***Objective Function*** will be as follow:

$$
\begin{aligned}
Objective\ Function \rightarrow F(&Target, w_{11}, w_{12}, w_{13}, w_{14}, b_{11}, b_{12}, b_{21}, x_1, x_2) \\
&= CrossEntropy(Target, Softmax(w_{21} \cdot Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 \\
&+ b_{11}) + w_{22} \cdot Sigmoid(w_{12} \cdot x_1 + w_{14} \cdot x_2 + b_{12}) + b_{21}))
\end{aligned}
$$

The ***gradient*** of Objective function on each ***weight*** and ***bias*** will be as follow:

$$
\nabla w_i = \frac{\partial F(Target, w_{11}, w_{12}, w_{13}, w_{14}, b_{11}, b_{12}, b_{21}, x_1, x_2)}{\partial w_i}
$$

$$
\nabla b_i = \frac{\partial F(Target, w_{11}, w_{12}, w_{13}, w_{14}, b_{11}, b_{12}, b_{21}, x_1, x_2)}{\partial b_i}
$$

The derivative of **Softmax** and **Cross-entropy** function is elegant:

$$
\frac{\partial CrossEntropy(\boldsymbol{y}, Softmax(x_i))}{\partial_{x_i}} = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} - y_j = softmax\ (\boldsymbol{x}_i) - y_j
$$

The equation of derivative of **Sigmoid** activation function is listed below:

$$
\frac{d(Sigmoid\ (x))}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = Sigmoid\ (x)(1 - Sigmoid\ (x))
$$

After combining the above equations, the gradients of weights and biases between hidden layer and output layer can be calculated:

$$
\begin{aligned}
\nabla w_{21} = &\frac{\partial F\big(Target, w_{11}, w_{12}, w_{13}, w_{14}, b_{11}, b_{12}, b_{21}, x_1, x_2\big)}{\partial w_{21}} \\
&= \frac{\partial CrossEntropy\big(Target, Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})\big)}{\partial_{w_{21}}} \\
&= \frac{\partial CrossEntropy\big(Target, Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})\big)}{\partial(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})} \\
&\cdot \frac{\partial(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})}{\partial_{w_{21}}} \\
&= (Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21}) - Target) \cdot x_1' \\
&= (Y - Target) \cdot {x_1}'
\end{aligned}
$$

$$
\begin{aligned}
\nabla b_{21} = &\frac{\partial F\big(Target, w_{11}, w_{12}, w_{13}, w_{14}, b_{11}, b_{12}, b_{21}, x_1, x_2\big)}{\partial b_{21}} \\
&= \frac{\partial CrossEntropy\big(Target, Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})\big)}{\partial_{b_{21}}} \\
&= \frac{\partial CrossEntropy\big(Target, Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})\big)}{\partial(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})} \\
&\cdot \frac{\partial(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})}{\partial_{b_{21}}} = Y - Target
\end{aligned}
$$

The gradients of weights and biases between hidden layer and input layer can be calculated as follow:

$$\nabla w_{11} = \frac{\partial F(Target, w_{11}, w_{12}, w_{13}, w_{14}, b_{11}, b_{12}, b_{21}, x_1, x_2)}{\partial w_{11}}$$

$$= \frac{\partial CrossEntropy\big(Target, Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})\big)}{\partial_{x_1'}}$$

$$\cdot \frac{\partial Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}{\partial_{w_{11}}}$$

$$= \frac{\partial CrossEntropy\big(Target, Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})\big)}{\partial(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})}$$

$$\cdot \frac{\partial(w_{21} \cdot Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11}) + w_{22} \cdot x_2' + b_{21})}{\partial Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}$$

$$\cdot \frac{\partial Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}{\partial_{(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}} \cdot \frac{\partial(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}{\partial_{w_{11}}}$$

$$= (Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21}) - Target) \cdot w_{21} \cdot Sigmoid'(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11}) \cdot x_1$$

$$= (Y - Target) \cdot w_{21} \cdot Sigmoid'(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11}) \cdot x_1$$

$$\nabla b_{11} = \frac{\partial F(Target, w_{11}, w_{12}, w_{13}, w_{14}, b_{11}, b_{12}, b_{21}, x_1, x_2)}{\partial b_{11}}$$

$$= \frac{\partial CrossEntropy\big(Target, Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})\big)}{\partial_{x_1'}}$$

$$\cdot \frac{\partial Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}{\partial_{b_{11}}}$$

$$= \frac{\partial CrossEntropy\big(Target, Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})\big)}{\partial(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21})}$$

$$\cdot \frac{\partial(w_{21} \cdot Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11}) + w_{22} \cdot x_2' + b_{21})}{\partial Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}$$

$$\cdot \frac{\partial Sigmoid(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}{\partial_{(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}} \cdot \frac{\partial(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})}{\partial_{b_{11}}}$$

$$= (Softmax(w_{21} \cdot x_1' + w_{22} \cdot x_2' + b_{21}) - Target) \cdot w_{21} \cdot Sigmoid'(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})$$

$$= (Y - Target) \cdot w_{21} \cdot Sigmoid'(w_{11} \cdot x_1 + w_{13} \cdot x_2 + b_{11})$$

Finally update weights $w_{new} = w_{old} + \mu \cdot (-\nabla w_{old})$, and biases $b_{new} = b_{old} + \mu \cdot (-\nabla b_{old})$, in which μ is learning rate.

Optimization Techniques

Momentum

Sometimes the gradient descent algorithm may lead the model into a poor local minimum, the figure below shows a possible local minimum situation.

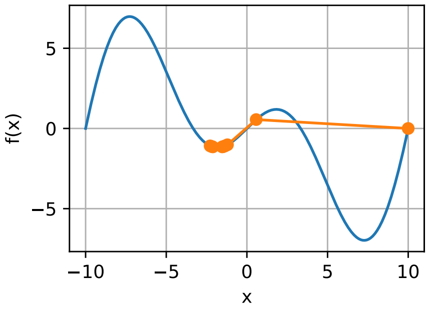

Figure 4 Local Minima

A possible solution is to add a momentum which store the last update information. The parameter updating process will be different from previous:

$$w_{new} = w_{old} + \mu \cdot (-\nabla w_{old}) + m\Delta w_{old}$$

$$b_{new} = b_{old} + \mu \cdot (-\nabla b_{old}) + m\Delta b_{old}$$

In which $\boldsymbol{m}$ is the hyper-parameter of momentum and $\Delta \boldsymbol{w_{old}}, \Delta \boldsymbol{b_{old}}$ are the updates of weights and biases in last iteration.

Adaptive Learning Rate

Another problem is the learning rate. When the learning rate is set too small, the training will be too slow, however, when the learning rate is too large, the system may be not able to converge. The following figure shows the situation when the learning rate is too large:

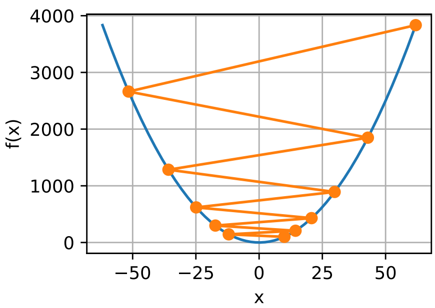

Figure 5 Too Large Learning Rate

A possible solution is to set adaptive learning. The initial learning rate could be set a little bit larger value and detect the error during the training. When the error stop decreases, make the system automatically decrease the learning rate.

Overfitting

The BP neural network is very likely to have the overfitting problem. Which means that the model overfits the training dataset but performance poor on test dataset and on real data. The main reasons can be summarized as the following three:

1. The model is too complex, and the training dataset is too simple.
2. There are noise data in the training dataset, and the model learns the noise data characteristics, which leads to the deviation of normal data processing.
3. Too many learning iterations, the model overfitted the training sample of non-representative data.

The possible solutions include early stopping, L2 regularization and using simple model. In this report we focus on the first one, which means that we stop the training process when the error is smaller than a specific value.

Multilayer BP Neural Network Implementation

Model Initialization

Initialization of the model, details are included in the **comments** of the code.

```
class NN:
    #initialize the model
    #hidden should be passed a list
    #e.g. hidden=[15,8] means model will have two hidden layers, first hidden layer has 15 nodes and second one has 8 nodes
    #attributes means the number of attributes
    #classes means the number of classes in the cllasification tasks
    #activation means the activation function, mush be "ReLU" or "Sigmoid"
    def __init__(self, hidden, attributes, classes, activation):
        #Size store the number of nodes in each layer
        self.size = [attributes] + hidden + [classes]
        #value in each node in each layer
        self.nodes = []
        #value of each weight
        self.weights = []
        #value of each bias
        self.bias = []
        #momentum is used to help get rid of local minima
        #value of momentum of each weight
        self.momentums_w = []
        #value of momentum of each bias
        self.momentums_b = []
        #number of classes to classified
        self.classes = classes
        #activation function for hidden layers ("ReLU" or "Sigmoid")
        self.activation=activation
        #expand the size of nodes in each layer
        for layer in range(len(self.size)):
            self.nodes.append([0.0] * self.size[layer])
        #expand the size of bias in each layer
        for layer in range(1, len(self.size)):
            self.bias.append([0.0] * self.size[layer])
            self.momentums_b.append([0.0] * self.size[layer])
        #initialize biases with a random value between -1 and 1
        for layer in range(1, len(self.size)):
            for cell in range(self.size[layer]):
                self.bias[layer-1][cell] = random.uniform(-1, 1)
        #expand the size of weights in each layer
        for layer in range(len(self.size) - 1):
            matrix = []
            for _ in range(self.size[layer]):
              matrix.append([0.0] * self.size[layer + 1])
            self.weights.append(copy.deepcopy(matrix))
            self.momentums_w.append(copy.deepcopy(matrix))
        #initialize weights with a random value between -1 and 1
        for layer in range(len(self.size) - 1):
            for m in range(self.size[layer]):
                for n in range(self.size[layer + 1]):
                    self.weights[layer][m][n] = random.uniform(-1, 1)
```

Define the ReLU and Sigmoid activation function

```python
    #define the activation function(Sigmoid) for hidden layers
    def Sigmoid(self, input):
        if input >= 0:
            return 1.0 / (1.0 + math.exp(-input))
        else:
            return math.exp(input) / (1.0 + math.exp(input))

    #define the activation function(ReLU) for hidden layers
    def ReLU(self, input):
      if input > 0:
        return input
      else:
        return 0

    def Activation(self, input):
      if self.activation == "ReLU":
        return self.ReLU(input)
      if self.activation == "Sigmoid":
        return self.Sigmoid(input)
```

Define the derivative of the activation functions (ReLU and Sigmoid)

```python
    #define the Derivative of activation function(Sigmoid) for hidden layers
    def Derivative_Sigmoid(self, input):
        return input * (1 - input)

    #define the Derivative of activation function(ReLU) for hidden layers
    def Derivative_ReLU(self, input):
        if input>0:
          return 1
        else:
          return 0

    def Derivative_Activation(self,input):
      if self.activation=="ReLU":
        return self.Derivative_ReLU(input)
      if self.activation=="Sigmoid":
        return self.Derivative_Sigmoid(input)
```

Define the Softmax function and loss function and their derivative

```python
    #define the activation function(Softmax) for output layers
    def Softmax(self, inputs):
        sum = 0
        for i in inputs:
            sum += math.exp(i)
        return [math.exp(input)/sum for input in inputs]

    #define the loss function(cross entropy) for output
    def Loss(self, outputs, labels):
        results = []
        for i in range(len(labels)):
            results.append(-labels[i]*math.log(1e-6+outputs[i])/len(labels))
        return results

    #define the derivative for output layers (d(CrossEntropy(target,SoftMax(x))/dx)
    def Derivative_Loss(self, output, label):
        return output-label
```

The forward propagation processes

Lines 118-119: Transmit input data to input nodes
Lines 121-128: Calculate outputs of hidden layers' nodes
Lines 130-136: Calculate outputs of output layer nodes

```
116     #forward propagate
117     def Forward(self, inputs):
118         for i in range(self.size[0]):
119             self.nodes[0][i] = inputs[i]
120         last_layer = len(self.size)-1
121         for layer in range(1, last_layer):
122             for node in range(self.size[layer]):
123                 total = 0.0
124                 for i in range(self.size[layer - 1]):
125                     total += self.nodes[layer - 1][i] * \
126                         self.weights[layer - 1][i][node]
127                 total += self.bias[layer-1][node]
128                 self.nodes[layer][node] = self.Activation(total)
129 
130         for node in range(self.size[last_layer]):
131             total = 0.0
132             for i in range(self.size[last_layer - 1]):
133                 total += self.nodes[last_layer - 1][i] * self.weights[last_layer - 1][i][node]
134             total += self.bias[last_layer-1][node]
135             self.nodes[last_layer][node] = total
136         self.nodes[last_layer] = self.Softmax(self.nodes[last_layer])
137         #return the result
138         return self.nodes[-1]
```

The backward propagation processes

Lines 157-172: Calculate parts of the gradients that can be shared with multiple nodes and store in a list called **deltas**

Lines 174-185: Calculate **gradients** and **momentums** corresponding to each weights and biases and update them.

```
150     #back propagate
151     def Back(self, input, label, learn_r, mom):
152 
153         self.Forward(input)
154         deltas = []
155 
156         #error back propagate
157         for layer in reversed(range(1, len(self.size))):
158             if layer == len(self.size) - 1:
159                 deltas.insert(0,[0.0] * self.size[layer])
160                 for i in range(self.size[layer]):
161                     # Derivative of error for each outputs
162                     error = self.Derivative_Loss(self.nodes[layer][i], label[i])
163                     deltas[0][i] = error
164             else:
165                 deltas.insert(0,[0.0] * self.size[layer])
166                 # transmit the Derivative of error to each node
167                 for i in range(self.size[layer]):
168                     error = 0.0
169                     for j in range(self.size[layer + 1]):
170                         error += deltas[1][j] * self.weights[layer][i][j]
171                     deltas[0][i] = self.Derivative_Activation(self.nodes[layer][i]) * error
172 
173         #Update weights
174         for layer in reversed(range(1, len(self.size))):
175             for i in range(self.size[layer - 1]):
176                 for j in range(self.size[layer]):
177                     delta_w = - deltas[layer - 1][j] * self.nodes[layer - 1][i] * learn_r + mom * self.momentums_w[layer - 1][i][j]
178                     self.weights[layer - 1][i][j] += delta_w
179                     self.momentums_w[layer - 1][i][j] = delta_w
180         #Update biases
181         for layer in reversed(range(1, len(self.size))):
182             for i in range(self.size[layer]):
183                 delta_b = - deltas[layer - 1][i] * learn_r + mom * self.momentums_b[layer - 1][i]
184                 self.bias[layer - 1][i] += delta_b
185                 self.momentums_b[layer - 1][i] = delta_b
186 
187         #calculate new error
188         self.Forward(input)
189         error = 0.0
190         result = self.Loss(self.nodes[len(self.nodes) - 1], label)
191         for i in range(len(label)):
192             error += result[i]
193         return error
```

Define the training process

Variable: max_round=max iterations, **stay**=number of iterations that error stays at the same level
Line 194: Shuffle the input samples
Lines 196-200: if error stays at the same level in last 10 iterations, reduce the learning rate
Lines 201-204: training the model
Lines 208-213: update **stay** according to the error in this iteration

```
186     #Define the training process
187     def Train(self, inputs, labels, max_round, learn_r, mom, min_error):
188         error = 0
189         stay = 0
190         last_error = 10e10
191         print("Learn Rate--->{0}".format(learn_r))
192         for i in range(max_round):
193             error = 0
194             sample = random.sample(range(0, len(inputs)), len(inputs))
195             #adaptive learning rate
196             if stay >= 10:
197                 learn_r = learn_r * 0.9
198                 stay = 0
199                 print()
200                 print("Update Learn Rate--->{0}".format(learn_r))
201             for j in sample:
202                 label = labels[j]
203                 input = inputs[j]
204                 error += self.Back(input, label, learn_r, mom)
205             error /= len(sample)
206             print("Training No.{: <5d}  Error:{: .10f}".format(i+1, error))
207 
208             if error > last_error * 1.01:
209                 stay += 10
210             elif error > last_error * 0.99:
211                 stay += 1
212             else:
213                 stay = 0
214 
215             last_error = error
216             #early stop when error<min_error
217             if error < min_error:
218                 print("Training End!      Error:{: .10f}".format(error))
219                 print()
220                 return
221         print("Training end!", "Error=", error)
222         print()
223         return
224 
225     #the whole training process, one-hot encoding labels, training
226     def fit(self, cases, labels, lr=0.1, max_iter=1000, mom=0.1, min_error=1e-3):
227         start = time.time()
228         print("Start training "+str(len(labels))+" samples...")
229         one_hot_labels = self.one_hot(labels)
230         self.Train(cases, one_hot_labels, max_iter, lr, mom, min_error)
231         end = time.time()
232         print("Elapsed time:" + str(end - start))
233         # return [self.size, self.nodes, self.weights, self.bias]
```

One-hot encoding

```
    #one-hot encode the label
    def one_hot(self, input):
        one_hot_labels = []
        for i in range(len(input)):
            label = []
            for j in range(self.classes):
                label.append(0)
            label[input[i]] = 1
            one_hot_labels.append(label)
        return one_hot_labels
```

Define the prediction function

```python
    #predict multiple inputs
    def predict(self, inputs):
        outputs = []
        for case in range(len(inputs)):
            result = self.Forward(inputs[case])
            outputs.append(result.index(max(result)))
        return outputs
```

Define the test function to verify the model

```python
    #verify the training results
    def test(self, cases, labels):
        right = 0
        print("Testing "+str(len(labels))+" samples...")
        print("Sample    Correct      Prediction      Label")
        results=self.predict(cases)
        for case in range(len(cases)):
            result = results[case]
            label = labels[case]
            if result == label:
                right += 1
                print("No.{:<5d}    *           {:=4d}         {:=4d}".format(case+1, result, label))
            else:
                print("No.{:<5d}                {:=4d}         {:=4d}".format(case+1, result, label))
        print("Accuracy : {: .2f}%".format(100 * right / len(cases)))
        print()
```

Model Evaluation

The model is evaluated on **iris** Dataset and UCI ML hand-written **digits** Dataset (Dua and Graff, 2017).

Data Preparation

The **iris** Dataset has 150 samples divided into 3 classes. Each sample has 3 positive numerical attributes. All attributes are transformed by **sklearn.standardscaler** function:

$$For\ each\ value\ x_i\ in\ each\ attribute\ X{:}\ x_i = (x_i - \mu)/s$$

In which μ =mean of the attribute and s =standard deviation of the attribute.

The UCI ML hand-written **digits** Dataset has 1797 samples divided into 10 classes. Each sample has 64 integer attributes between 0 and 16. All attributes are also transformed by **sklearn.standardscaler** function.

Experiment Design and Evaluation

For one round of test, the dataset will firstly be shuffled. Then the first 80% will be taken as the training samples and the rest 20% will be used to test the model. The hyper parameter settings are: hidden=[15,8], activation="Sigmoid", lr=0.1 (learning rate), max_iter=100, mom=0.1 (momentum hyper parameter), min_error=1e-3.

The test function of the model will output each prediction and target and then the overall accuracy rate.

Evaluation Results

Iris Dataset

Training process:

```
Start training 120 samples...
Learn Rate--->0.1
Training No.1       Error: 0.2574107662
Training No.2       Error: 0.1497464124
Training No.3       Error: 0.1095232992
Training No.4       Error: 0.0846160388
Training No.5       Error: 0.0664771507

…………..

Update Learn Rate--->0.006461081889226681
Training No.95      Error: 0.0149512803
Training No.96      Error: 0.0148371067
Training No.97      Error: 0.0148187776
Training No.98      Error: 0.0149312398
Training No.99      Error: 0.0149261009
Training No.100     Error: 0.0148450311
Training End!       Error: 0.0148450311
Elapsed time: 6.279797554016113
```

The training stopped at the max iteration and the final average error is 0.014845. The learning rate decreases from 0.1 to 0.00646.

Results:

```
Testing 30 samples...
Sample    Correct    Prediction    Label
No.1        *            2           2
No.2        *            2           2
No.3        *            1           1
No.4                     2           1
No.5        *            2           2

………..

No.28       *            2           2
No.29       *            0           0
No.30       *            1           1
Accuracy :   96.67%
```

On the 30 test samples, the accuracy rate is 96.67%

UCI ML hand-written **digits** Dataset

Training process:

Start training 1438 samples...
Learn Rate--->0.1

Training No.1	Error: 0.0922216482
Training No.2	Error: 0.0192242689
Training No.3	Error: 0.0091077559
Training No.4	Error: 0.0058665620
Training No.5	Error: 0.0041472968
Training No.6	Error: 0.0031858003
Training No.7	Error: 0.0025529499
Training No.8	Error: 0.0021964761
Training No.9	Error: 0.0018219710
Training No.10	Error: 0.0015851523
Training No.11	Error: 0.0013669221
Training No.12	Error: 0.0012268077
Training No.13	Error: 0.0011084607
Training No.14	Error: 0.0010010743
Training No.15	Error: 0.0009269276
Training End!	Error: 0.0009269276

Elapsed time: 62.10514640808105

The training stopped on error <min error at No. 15 iteration.

Results:

Testing 359 samples...

Sample	Correct	Prediction	Label
No.1	*	8	8
No.2	*	9	9
No.3	*	9	9
No.4	*	3	3
No.5	*	5	5

…………..

No.355	*	4	4
No.356	*	6	6
No.357	*	9	9
No.358	*	7	7
No.359	*	9	9

Accuracy : 96.10%

On the 359 test samples, the accuracy rate is 96.10%

Conclusion

In this implementation, a multilayer BP neural network is constructed. Users can define the number of hidden layers and number of nodes in each layer, and choose the activation function between “ReLU” and “Sigmoid” for hidden layers. To solve the local minima problem, the momentum is integrated into the model. To solve the overfitting problem, the model uses an early stop solution. In addition, the model can also automatically change the learning rate to fit the current learning process. On the two datasets (iris dataset and UCI ML hand-written digits Dataset), the accuracy rates of the model are 96.67% and 96.10% respectively. The future work could focus on integrating L2 regularization into the model to better solve the overfitting problem.

Reference

Dheeru Dua and Casey Graff. 2017. UCI machine learning repository.

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexan-der J. Smola. 2021. Dive into deep learning.arXivpreprint arXiv:2106.11342.