Deep Learning Framework Modules Documentation

Submitted By
Ahmed Sameh Abd El Hamid Shahed
Ahmed Yehia Ahmed Mohammed
Mohammed Emad Mahmoud Abd El Hamid
Mohammed Amr Ahmed Taha Mansi
Mohammed Amr Mohammed Hassan
Mohammed Hussien Mostafa Masoud
Mohammed Khaled Rashad Mohammed
Mohammed Khaled Mohammed Sayed El Khawas

January 24, 2021

1 Modules

1.1 DataPreProcessing

DataPreProcessing module is used in manipulating data to make it suitable to be used by the framework.

1.1.1 get_data(path, label_path = "", shuffle = False)

Used to get feature matrix and label vector out of files.
The feature matrix is a numpy array with dimensions (N X D), where N: number of samples and D: number of features in every sample.
The label vector is a numpy array with dimensions (N X 1), where N: number of samples.

- Parameters
  1. path: {'string'} The path of the file that contains the features and labels.
  2. label_path: {'string'} The path of labels file if label is not included in first path.
  3. shuffle: {'bool'} If True the rows will be rearranged randomly.
- Return: {'tuple'} of 2 items
  1. Feature Matrix: {'numpy.array'}
  2. Label Vector: {'numpy.array'}
- Limitations
  1. You must input label_path if there are no labels data in first path.
  2. Label column name must be 'Label' and can't be anything else.

1.1.2 normalize(matrix)

Used to normalize input matrix, The returned matrix has the same dimensions as the input.

- Parameters
  1. matrix: {'numpy.array'} Unormalized matrix.
- Return: {'numpy.array'} normalized matrix.

1.1.3 split_data(X, label)

Used to split data into training data set with it's training labels and test data set with it's test labels.
The data is split with splitting ratio 75% training data and 25% testing data.

- Parameters
  1. X: {'numpy.array'} The feature matrix.
  2. label: {'numpy.array'} The labels vector.
- Return: {'tuple'} of 4 items
  1. Training Feature Matrix: {'numpy.array'}
  2. Test Feature Matrix: {'numpy.array'}
  3. Training Label vector: {'numpy.array'}
  4. Test Label Vector: {'numpy.array'}

1.2 Saving and Importing Weights

1.2.1 save_weights(layer_arr)

Save a 2d array into csv file.

- Parameters
  1. layer_arr: {numpy_array} The array which holds numbers.
- Shape
  1. layer_arr: numpy array of shape (N, m) N: number of weights
- Return: {'void'}
- Example: save_weights(np.array([[1,2,3,4,56,9,7],[3,4,6,7,8]]))

1.2.2 import_saved_weights()

import a 2d array from csv file.

- Parameters
  1. model: {'model'} The structured model to be trained.
- Return: {'numpy.array'} The shape of the array is (N, m) where N: number of weights
- Example: weights = import_saved_weights()

1.3 Visualization

1.3.1 __init__(self, name="the graph", line1="loss", line2="percision")

Create a graph with a specific name and line names if there is two lines.

- Parameters
  1. name: {'string'} A string holds the graph name.
  2. line1: {'string'} A string holds the first line name.
  3. line2: {'string'} A string holds the second line name.
- Example: p1 = visualization("graph_title","first_line","seconde_line")

1.3.2 add_point_to_graph(self,new_value3,it,epoch)

Update the graph by adding the new point to it and print it and check if the final step reached we save the figure into picture and print it.

- Parameters
  1. new_value3: {'int'} The new value to be added to the graph.
  2. it: {'int'} The number of the iteration.
  3. epoch: {'int'} The iteration witch we stop at.
- Example: p1.add_point_to_graph(5,1,100)

1.3.3 add_two_points_to_graph(self,new_value1,new_value2)

Update the graph by adding two new points to it then print it.

- Parameters
  1. new_value1: {'int'} A new value to be added to the graph.
  2. new_value2: {'int'} A new value to be added to the graph.
- Example: p1.add_two_points_to_graph(10,4)

1.4 Activations

1.4.1 sigmoid(Z)

Calculate the sigmoid values and derivative of the sigmoid of the inputs.
based on the formulas:
A = 1/(1+e^(-Z))
A_dash = A(1-A)

- Parameters
  1. Z: {'numpy.array'} Input matrix.

- Shape

  1. Z (m, k) m: number of samples , k: number of neurons in current layer.

- Return: {'tuple'} of 2 items.

  1. A: {'numpy.array'} The values of sigmoid of the input, with shape (m,k) m: number of samples , k: number of neurons in current layer.
  2. A_dash: {'numpy.array'} The values of the derivative of the sigmoid related to the input, with shape (m,k) m: number of samples , k: number of neurons in current layer

- Example: result,result_der = sigmoid(np.array([4,1,-5,3]))

1.4.2 tanh(Z)

Calculate the tanh values and derivative of the tanh of the inputs.
based on the formulas:
A = (e$\hat{}$(Z)-e$\hat{}$(-Z))/(e$\hat{}$(Z)+e$\hat{}$(-Z))
A_dash = 1-A$\hat{2}$

- Parameters

  1. Z: {'numpy.array'} Input matrix.

- Shape

  1. Z (m, k) m: number of samples , k: number of neurons in current layer.

- Return: {'tuple'} of 2 items.

  1. A: {'numpy.array'} The values of tanh of the input, with shape (m,k) m: number of samples , k: number of neurons in current layer.
  2. A_dash: {'numpy.array'} The values of the derivative of the tanh related to the input, with shape (m,k) m: number of samples , k: number of neurons in current layer

- Example: result,result_der = tanh(np.array([4,1,-5,3]))

1.4.3 relu(Z)

Calculate the relu values and derivative of the relu of the inputs :
based on the formulas:
A = Z if Z ¿= 0 otherwise 0
A_dash = 1 if Z ¿= 0 otherwise 0

- Parameters

1. Z: {'numpy.array'} Input matrix.

- Shape

1. Z (m, k) m: number of samples , k: number of neurons in current layer.

- Return: {'tuple'} of 2 items.

1. A: {'numpy.array'} The values of relu of the input, with shape (m,k) m: number of samples , k: number of neurons in current layer.
2. A_dash: {'numpy.array'} The values of the derivative of the relu related to the input, with shape (m,k) m: number of samples , k: number of neurons in current layer

- Example: result,result_der = relu(np.array([4,1,-5,3]))

1.4.4 softmax(A,Y)

Calculate the softmax values and derivative of the softmax of the inputs.
based on the formulas:
A[i] = e^(Z[i])/sum(e^(Z[j])) j: from 1 to number of neurons in output layer
A_dash = 1 , as it is included in the softmax loss function

- Parameters

1. Z: {'numpy.array'} Input matrix.

- Shape

1. Z (m, k) m: number of samples , k: number of neurons in current layer.

- Return: {'tuple'} of 2 items.

1. A: {'numpy.array'} The values of softmax of the input, with shape (m,k) m: number of samples , k: number of neurons in current layer.
2. A_dash: 1

- Example: result,result_der = softmax(np.array([4,1,-5,3]))

1.5 Losses

1.5.1 mse(A,Y)

Calculate the mean square loss and derivative of the mean sqaure loss between label and prediction.
based on the formulas:
Loss = (1/2m)*(sum((A-Y)^2))
Loss_dash = (1/m)* (A-Y)

- Parameters
  1. A: {'numpy.array'} The activation output of the output layer.
  2. Y: {'numpy.array'} The labels of the examples.
- Shape
  1. A (m, k) m: number of samples , k: number of neurons in current layer.
  2. Y (m, k) m: number of samples , k: number of neurons in current layer.
- Return: {'tuple'} of 2 items.
  1. Loss: {'numpy.array'} The values of loss of the inputs, with shape (m,k) m: number of samples , k: number of neurons in current layer.
  2. Loss_dash: {'numpy.array'} The values of the derivative of the loss related to the inputs, with shape (m,k) m: number of samples , k: number of neurons in current layer
- Example: result,result_der = mse(np.array([4,1,-5,3]),np.array([0,0,1,0]))

1.5.2 nll(A,Y)

Calculate the negative log likelihood loss and derivative of negative log likelihood loss between label and prediction :
based on the formulas:
Loss = -log(prediction[yi]),yi is the index of the correct label
Loss_dash = -1/Y

- Parameters
  1. A: {'numpy.array'} The activation output of the output layer.
  2. Y: {'numpy.array'} The labels of the examples.
- Shape
  1. A (m, k) m: number of samples , k: number of neurons in current layer.
  2. Y (m, k) m: number of samples , k: number of neurons in current layer.
- Return: {'tuple'} of 2 items.
  1. Loss: {'numpy.array'} The values of loss of the inputs, with shape (m,k) m: number of samples , k: number of neurons in current layer
  2. Loss_dash: {'numpy.array'} The values of the derivative of the loss related to the inputs, with shape (m,k) m: number of samples , k: number of neurons in current layer
- Example: result,result_der = nll(np.array([4,1,-5,3]),np.array([0,0,1,0]))

1.5.3 l1(A,Y)

Calculate the mean absolute loss and derivative of the mean absolute loss between label and prediction.
based on the formulas:
Loss = (1/m)*sum(—A-Y—)
Loss_dash = 1 if A-Y ¿ 0 otherwise -1

- Parameters
  1. A: {'numpy.array'} The activation output of the output layer.
  2. Y: {'numpy.array'} The labels of the examples.
- Shape
  1. A (m, k) m: number of samples , k: number of neurons in current layer.
  2. Y (m, k) m: number of samples , k: number of neurons in current layer.
- Return: {'tuple'} of 2 items.
  1. Loss: {'numpy.array'} The values of loss of the inputs, with shape (m,k) m: number of samples , k: number of neurons in current layer
  2. Loss_dash: {'numpy.array'} The values of the derivative of the loss related to the inputs, with shape (m,k) m: number of samples , k: number of neurons in current layer
- Example: result,result_der = l1(np.array([4,1,-5,3]),np.array([0,0,1,0]))

1.5.4 softmax(A,Y)

Calculate the softmax loss and derivative of the softmax between label and prediction.
based on the formulas:
Loss = -log(prediction[yi]),yi is the index of the correct label
Loss_dash[r] = -(1-yi) if r= yi otherwise yi

- Parameters
  1. A: {'numpy.array'} The activation output of the output layer.
  2. Y: {'numpy.array'} The labels of the examples.
- Shape
  1. A (m, k) m: number of samples , k: number of neurons in current layer.
  2. Y (m, k) m: number of samples , k: number of neurons in current layer.

- Return: {'tuple'} of 2 items.
  1. Loss: {'numpy.array'} The values of loss of the inputs, with shape (m,k) m: number of samples , k: number of neurons in current layer
  2. Loss_dash: {'numpy.array'} The values of the derivative of the loss related to the inputs, with shape (m,k) m: number of samples , k: number of neurons in current layer
- Example: result,result_der = softmax(np.array([4,1,-5,3]),np.array([0,0,1,0]))

1.6 Layer

Encapsulate Layers parameters.

1.6.1 __init__(self,size,activation='Identity')

Create a Layer with a specific size and activation function.

- Parameters
  1. size: {'tuple'} of 2 items, (input size, neuron count) respectively.
  2. activation: {'string'} Specify the applied activation function on layer. available activation functions: [Identity, ReLU, Sigmoid, Tanh, Softmax] Default: Identity.
- Shape
  1. size: numpy array of shape(N, m) N: number of features.
- Example: layer1 = nn.Layer(size=(3,5), activation='ReLU')

1.6.2 forward(self,inputs)

Calculate a forward propagation step:
based on the Linear formula: Y = activation(X.W + b).

- Parameters
  1. inputs: {'numpy.array'} A numpy array with the previous layer values (or the network inputs).
- Shape
  1. inputs: numpy array of shape (1, N) N: number of features.
- Return: {'numpy.array'} The shape of the array is (1, m), A forward propagation step value after applying activation function.
- Example
  layer = nn.Layer(size=(3,5), activation='ReLU')
  result = layer.forward(np.array([4,1,-5,3]))

1.6.3 __call__(self,inputs)

Calculate a forward propagation step over one layer:
based on the Linear formula: Y = activation(X.W + b).

- Parameters
  1. inputs: {'numpy.array'} A numpy array with the previous layer values (or the network inputs).
- Shape
  1. inputs: numpy array of shape (1, N) N: number of features.
- Return: {'numpy.array'} The shape of the array is (1, m), A forward propagation step value after applying activation function.
- Example
  layer = nn.Layer(size=(3,5), activation='ReLU')
  result = layer(np.array([4,1,-5,3]))

1.7 Model

Model class encapsulates Layers objects

1.7.1 __init__(self,*layers)

Create a Model with a specific number and type of Layers.

- Parameters
  1. layers: {'Layer'} Multi-valued parameter that holds one or more Layer object.
- Shape
  1. layers: (nn.Layer(N,m), nn.Layer(m,k), ... , nn.Layer(c,1)).
- Example
  model = nn.Model(Layer(size=(3,5), activation='ReLU'),
  Layer(size=(5,10), activation='ReLU'), Layer(size=(10,6), activation='ReLU'),
  Layer(size=(6,1), activation='ReLU'))

1.7.2 forward(self,inputs)

Calculate a forward propagation step over the whole model:
based on the Linear formula:[A = activation(Layer(x)), B = activation(Layer(A)), . . . Y = activation(Layer(F))]

- Parameters

1. inputs: {'numpy.array'} A numpy array with the network inputs.

- Shape

1. inputs: numpy array of shape (1, N) N: number of features.

- Return: {'numpy.array'} The shape of the array is (1, m), A forward propagation step value after applying activation function.

- Example
model = nn.Model(Layer(size=(3,5), activation='ReLU'), Layer(size=(5,1), activation='ReLU'))
model.forward(np.array([1,6,-2]))

1.7.3 __call__(self,inputs)

Calculate a forward propagation step over the whole model:
based on the Linear formula:[A = activation(Layer(x)), B = activation(Layer(A)), . . . Y = activation(Layer(F))].

- Parameters

1. inputs: {'numpy.array'} A numpy array with the network inputs.

- Shape

1. inputs: numpy array of shape (1, N) N: number of features.

- Return: {'numpy.array'} The shape of the array is (1, m), A forward propagation step value after applying activation function.

- Example
model = nn.Model(Layer(size=(3,5), activation='ReLU'), Layer(size=(5,1), activation='ReLU'))
model(np.array([1,6,-2]))

1.7.4 fit(self,dataset_input,label,optimization_type,loss_type,alpha,epoch,graph_on = False)

Executing the learning process for a given dataset.

- Parameters

1. dataset_input: {'numpy.array'} The source inputs
2. label: {'numpy.array'} The correct label for each example
3. optimizatio_type: {'string'} The required learning optmization type
4. loss_type: {'string'} The required output loss function to use
5. alpha: {'float} The required learning rate
6. epoch: {'int'} The required number of iterations in learning process

7. graph_on: {'bool'} A boolean typed value to visualize the process

- Shape: The shape has no changes, it just changes the current values for learning
- Return: {'void'}
- Example: model.fit(X_train,label_train,'SGD','MSE',alpha = 0.0001,epoch = 50,graph_on = True)

1.7.5 evaluate(self,test_x,test_y,metric='Accuracy',beta=1.0)

Calculate the evaluation matrices for the testing data set.

- Parameters
  1. test_x: {'numpy.array'} A numpy array with the network testing inputs.
  2. test_y: {'numpy.array'} A numpy array with network testing true labels.
  3. metric: {'list'} Metric name(s) as string or list of strings.
     avliable metrics:[Accuracy, Confusion matrix, Precision, Recall, F1 score, FBeta score], Default: Accuracy.
  4. beta: {'float'} A hyperparameter value used to calculate FBeta score.
- Shape
  1. input:
     (a) test_x: (K, N) K:number of testing samples, N: number of features.
     (b) test_y: (K, m) K:number of testing samples, m: number of neurons at output.
  2. output: The specified metric value(s).
     beside storing all metrics in model variables as follows:
     self.accuracy [0-1] value
     self.confusion_matrix (2,2) matrix
     self.recall [0-1] value
     self.precision [0-1] value
     self.f1_score [0-1] value
     self.fbeta_score [0-1] value
- Example
  P,R,F1 = model.evaluate(np.array([[1,6,-2],[3,9,12],[7,-3,4]]), np.array([[0],[1],[1]]), metric=['Precision','Recall','F1_score','FBeta_score'], beta=0.6)

1.7.6 save(self,path = "model.NND")

Save model Object on Disk.

- Parameters
  1. path: {'string'} Path to the model to be saved. Default: "model.NND"
- Example: model.save("my model.NND")

1.7.7 load(path)

Import loaded model from Disk.

- Parameters
  1. path: {'string'} Path to the model to be loaded.
- Return: {'Model'}
- Example: model = nn.load("my model.NND")

1.8 Optimization

1.8.1 sgd(model,alpha,sample,dloss)

makes one iteration on the given model using online optimization:
based on the Linear formula:
dl/dweights = d(loss)/d(activation_function n) * d(activation_function n)/d(prediction n) * d(prediction n)/d(Input n) ....* d(activation_function 1)/d(prediction 1)* d(prediction 1)/d(Input_dataset)

- Parameters
  1. model: {'model'} The structured model to be trained.
  2. alpha: {'float'} The learning rate.
  3. sample: {'numpy.array'} One sample from the dataset.
  4. dloss: {'numpy.array'} Derivative of the loss function by the activation function of the last layer.
- Shape
  1. model: Object from the model class.
  2. alpha: Scalar value.
  3. sample: numpy.array of shape (1 x number of features).
  4. dloss: numpy.array of shape (the same shape of the weights of the last layer).
- Return: {'void'} The function updates the weights and biases of the layers but doesn't return any thing.
- Example: sgd(model , 0.1 , np.array([1 , 2 , 3 , 4]) , np.array([6]))

1.8.2 batch(model,sample,dloss)

makes one iteration on the given model using batch optimization:
based on the Linear formula:
dl/dweights = d(loss)/d(activation_function n) * d(activation_function n)/d(prediction n) * d(prediction n)/d(Input n) ....* d(activation_function 1)/d(prediction 1)* d(prediction 1)/d(Input_dataset).

- Parameters
  1. model: {'model'} The structured model to be trained.
  2. sample: {'numpy.array'} One sample from the dataset.
  3. dloss: {'numpy.array'} Derivative of the loss function by the activation function of the last layer.
- Shape
  1. model: Object from the model class.
  2. sample: numpy.array of shape (1 x number of features).
  3. dloss: numpy.array of shape (the same shape of the weights of the last layer).
- Return: {'void'} the function accumilates the gredient of the loss wrt. weights and biases.
- Example: batch(model , np.array([1 , 2 , 3 , 4]) , np.array([6]))

1.8.3 norm(model, size_of_dataset)

This function used in the FIT function to get the norm of the LASTLAYER to campere to eipslon to stop the while loop.

- Parameters
  1. model: {'model'} The structured model to be trained.
  2. size_of_dataset: {'int'} Length of dateset to make normaliztion to data.
- Return: {'void'}

1.8.4 init_delta(model)

This function makes zeros of all weights_Grad(delta) matrix in begin of optimations of every layer.

- Parameters
  1. model: {'model'} The structured model to be trained.
- Return: {'void'}

1.8.5 update_weights_bias(model, alpha, size_of_dataset)

This function is rule to update weights of each layer according to that rule: Wi+1=Wi-Alpha*gard.

- Parameters
  1. model: {'model'} The structured model to be trained.
  2. alpha: {'float'} The learning rate.
  3. size_of_dataset: {'int'} The length of data set.
- Return: {'void'}