machine_learning_functions.py

```
import numpy as np
import random

class Leaky_RELU:
    def __init__(self, leak=0.01):
        self.leak = leak
        self.scalar_function = lambda x: self.leak*x if x < 0 else x
        self.scalar_derivative = lambda x: self.leak if x < 0 else 1.0

        self.vectorised_function = np.vectorize(self.scalar_function)
        self.vectorised_derivative = np.vectorize(self.scalar_derivative)

    def __call__(self, X):
        # return self.vectorised_function(X).reshape(-1, 1)
        return self.vectorised_function(X)

    def dX(self, X):
        return np.diag(
            # self.vectorised_derivative(X).squeeze()
            self.vectorised_derivative(X)
        )


class RELU(Leaky_RELU):
    def __init__(self):
        super().__init__(leak = 0.0)

class Linear_Activation_Funcion():
    def __call__(self, X):
        return X

    def dX(self, X):
        X_size = len(X)
        return np.eye(X_size)

class Sigmoid():
    def __init__(self):
        def exp(x):
            previous_term = 1
            result = 1
            for r in range(1, 10):
                previous_term += x/r
                result += previous_term
            return result

        self.scalar_function = lambda x: 1/(1+exp(x))
        # self.scalar_derivative = lambda x: self.scalar_function(x) * (1 -
self.scalar_function(x))

        self.last_X_cache = {}

        self.vectorised_function = np.vectorize(self.scalar_function)
        # self.vectorised_derivative = np.vectorize(self.scalar_derivative)
```

```python

    def calculate_and_cache(self, X):
        if not np.array_equal(X, self.last_X_cache.get("X")):
            self.last_X_cache["X"] = X
            self.last_X_cache["f(X)"] = self.vectorised_function(X)
            self.last_X_cache["f'(X)"] = np.diag(
                np.array(
                    [
                        x*(1-x)
                        for x in self.last_X_cache["f(X)"]
                    ]
                )
            )

    def __call__(self, X):
        self.calculate_and_cache(X)
        return self.last_X_cache["f(X)"]

    def dX(self, X):
        self.calculate_and_cache(X)
        return self.last_X_cache["f'(X)"]



# define a class for a layer
# this will represent a layer in a feed foreward neural network
class Layer_Transformation():
    def initalise_parameters(self):
        self.bias_dimensions = self.neurons
        self.wieghts_dimensions = (self.neurons, self.neurons_prev)

        # self.bias = np.zeros(self.bias_dimensions).reshape(-1, 1)
        # self.bias = np.zeros(self.bias_dimensions)
        # self.bias = np.full(self.bias_dimensions, 0.1)
        self.bias = np.full(self.bias_dimensions, 0.5)
        self.weights = np.random.uniform(-1, 1, self.wieghts_dimensions)

        self.activations = None
        self.weighted_sums = None
        self.activations_prev = None

    def __init__(self, num_neurons, num_neurons_previous_layer, activation_function) -> None:
        self.activation_func = activation_function
        self.neurons = num_neurons
        self.neurons_prev = num_neurons_previous_layer

        self.initalise_parameters()

    def get_activations(self, previous_activations):
        # assert isinstance(previous_activations, np.ndarray), repr(previous_activations)
        # assert previous_activations.shape == (self.neurons_prev,)

        self.activations_prev = previous_activations
        self.activations_prev = self.activations_prev if isinstance(self.activations_prev, np.ndarray) else np.array([self.activations_prev])

```

```python
        # @ is opetation for matrix multiplication
        self.weighted_sums = (self.weights @ self.activations_prev) + self.bias
        self.weighted_sums = self.weighted_sums if isinstance(self.weighted_sums, np.ndarray) else np.array([self.weighted_sums])

        # self.weighted_sums.reshape(-1, 1)
        self.activations = self.activation_func(self.weighted_sums)
        self.activations = self.activations if isinstance(self.activations, np.ndarray) else np.array([self.activations])


        # self.activations.reshape(-1, 1)
        return self.activations

    # derivative of activation with respect to weighted sum
    def dAdZ(self):
        assert not any(value is None for value in (self.weighted_sums, self.activations, self.activations_prev))
        return self.activation_func.dX(self.weighted_sums)

    # derivative of weighted sum with respect to weights matrix
    def dZdW(self):
        assert not any(value is None for value in (self.weighted_sums, self.activations, self.activations_prev))
        return self.activations_prev


    # redundant as it is always identity
    # # derivative of weighted sum with respect to bias vector
    # def dZdB(self):
    #     assert not any(value is None for value in (self.weighted_sums, self.activations, self.activations_prev))
    #     # eye function gives identity matrix
    #     return np.eye(self.bias_dimensions)

    # derivative of weighted sum with respect to previous activation
    def dZdAp(self):
        assert not any(value is None for value in (self.weighted_sums, self.activations, self.activations_prev))
        return self.weights

    def calculate_derivative(self, dcdA):
        # this function returns dcdW, dcdB and dcdAp

        # calculate relevant derivatives
        dAdZ = self.dAdZ()
        dZdW = self.dZdW()
        dZdAp = self.dZdAp()

        # apply the chain rule
        dcdZ = dAdZ @ dcdA
        dcdW = np.outer(dcdZ, dZdW)
        dcdB = dcdZ
        dcdAp = dZdAp.T @ dcdZ

        return dcdW, dcdB, dcdAp

```

```

    def update_weights(self, weights_change):
        # assert isinstance(weights_change, np.ndarray) or isinstance(weights_change, np.float64)
        # if isinstance(weights_change, np.ndarray):
        #     assert weights_change.shape == (self.neurons, self.neurons_prev)
        assert isinstance(weights_change, np.ndarray)
        assert weights_change.shape == (self.neurons, self.neurons_prev)

        self.weights += weights_change

    def update_bias(self, bias_change):
        # assert isinstance(bias_change, np.ndarray) or isinstance(bias_change, np.float64)
        # if isinstance(bias_change, np.ndarray):
        assert isinstance(bias_change, np.ndarray)
        assert bias_change.shape == (self.neurons,)

        self.bias += bias_change


    def get_weights(self):
        return self.weights

    def get_bias(self):
        return self.bias


class MSE():
    def __init__(self):
        self.diff = None
        self.n = None
    def __call__(self, P, Y):
        P = P if isinstance(P, np.ndarray) else np.array([P])
        Y = Y if isinstance(Y, np.ndarray) else np.array([Y])

        self.n = P.shape[0]
        self.diff = (P-Y)

        # here I will use dot product rather than transverse vectors which are not well supported in numpu :(
        return (1/self.n) * np.dot(self.diff, self.diff)

    def dP(self):
        assert not any(value is None for value in (self.diff, self.n))
        return (2/self.n) * self.diff


class FFN():
    def __init__(self, neurons_per_layer_list, activation_functions_list, cost_function):
        # input layer in addition to calculated layers
        assert len(neurons_per_layer_list) == len(activation_functions_list) +1
        self.num_transformation_layers = len(activation_functions_list)

        activation_functions_list = [
            Linear_Activation_Funcion() if activation_func is None else activation_func
            for activation_func in activation_functions_list
        ]
```

```python

        self.tranformation_layers = []
        previous_layer_neurons = neurons_per_layer_list[0]
        for neurons, activation_fuction in zip(neurons_per_layer_list[1:], activation_functions_list):
            self.tranformation_layers.append(
                Layer_Transformation(
                    neurons,
                    previous_layer_neurons,
                    activation_fuction
                )
            )
            previous_layer_neurons = neurons

        self.cost_function = cost_function

    def foreward_propagate(self, input_vector, expected_output_vector=None):
        # get network prediction
        layer_activations = input_vector
        for layer_transformation in self.tranformation_layers:
            layer_activations = layer_transformation.get_activations(layer_activations)

        if expected_output_vector is None:
            cost = None
        else:
            cost = self.cost_function(layer_activations, expected_output_vector)

        return layer_activations, cost

    def back_propogate(self, input_vector, expected_output_vector, do_fp=True):
        if do_fp:
            self.foreward_propagate(input_vector, expected_output_vector)

        param_grad_dict = {}

        dcdP = self.cost_function.dP()
        dcdA = dcdP
        for layer_num, layer in zip(
            range(self.num_transformation_layers, 0, -1),
            self.tranformation_layers[::-1]
        ):
            dcdW, dcdB, dcdAp = layer.calculate_derivative(dcdA)
            param_grad_dict[f"W{layer_num}"], param_grad_dict[f"B{layer_num}"] = dcdW, dcdB
            dcdA = dcdAp

        return param_grad_dict

    def update_parameters(self, parameter_chages):
        for param_name, param_chage in parameter_chages.items():
            layer_num = int(param_name[1])
            if param_name[0] == "W":
                self.tranformation_layers[layer_num-1].update_weights(param_chage)
            else:
                self.tranformation_layers[layer_num-1].update_bias(param_chage)

    def print_parameters(self):
```

```
        print("Parameters of network")
        for layer_i, layer in enumerate(self.tranformation_layers):
            print({f"W{layer_i+1}": layer.get_weights()})
            print({f"B{layer_i+1}": layer.get_bias()})




class Model():
    def __init__(self, FFN: FFN, data_set):
        self.FFN = FFN

        X_data, Y_data = data_set
        self.X_data = X_data
        self.Y_data = Y_data

        assert len(X_data) == len(Y_data)
        self.num_data_items = len(X_data)

        data_item_indexes = np.array(range(self.num_data_items))
        np.random.shuffle(data_item_indexes)

        test_train_ration = 0.8
        partition_index = int((self.num_data_items * test_train_ration) // 1)

        self.num_training_data_items = partition_index
        self.train_data_indexes = data_item_indexes[:partition_index]
        self.num_test_data_items = self.num_data_items - self.num_training_data_items
        self.test_data_indexes = data_item_indexes[partition_index:]

    def reset_model(self):
        self.FFN.initalise_parameters()

    def train_and_evaluate(self, learning_rate, epochs, batch_size):
        # beak up training data into many batches based on batch size and epochs
        batches = []
        total_batches = self.num_training_data_items // batch_size
        # purposly iterates total_batches times, bit after last partition discarded
        partition_indicies = [(batch_size-1)*partition_num for partition_num in range(1, total_batches+1)]
        for _ in range(epochs):
            np.random.shuffle(self.train_data_indexes)
            previous_partition_index = 0
            for partition_index in partition_indicies:
                batches.append(self.train_data_indexes[previous_partition_index:partition_index])
                previous_partition_index = partition_index



            # for each batch in list
            old_loss = None
            for batch_num, batch in enumerate(batches):
                # repeatedly complete back propagation
                total_param_cost_gradients = {}
                total_cost = 0
                for data_item_index in batch:
```

```
                    X, Y = self.X_data[data_item_index], self.Y_data[data_item_index]

                    # _, cost = self.FFN.foreward_propagate(X, Y)
                    _, cost = self.FFN.foreward_propagate(
                        input_vector=X,
                        expected_output_vector=Y
                    )

                    total_cost += cost

                    param_gradients = self.FFN.back_propogate(X, Y, False)
                    for param_name in param_gradients.keys():
                        if total_param_cost_gradients.get(param_name) is None:
                            total_param_cost_gradients[param_name] = param_gradients[param_name]
                        else:
                            total_param_cost_gradients[param_name] += param_gradients[param_name]

                new_loss = total_cost / batch_size
                if old_loss is not None:
                    loss_change = new_loss - old_loss
                    # print(f"batch {batch_num}: Loss change was {loss_change:.10f}")
                old_loss = new_loss


                # approximate parapeter gradients with repect to loss through mean of batch
                # use SGD algorithm to updata parameters
                parameter_loss_grads = {}
                parameter_chages ={}
                for param_name, total_cost_grad in total_param_cost_gradients.items():
                    parameter_loss_grads[param_name] = total_cost_grad / batch_size
                    parameter_chages[param_name] = (-learning_rate) * parameter_loss_grads[param_name]


                # print(f"Paramters were")
                # print((
                #     self.FFN.tranformation_layers[0].weights,
                #     self.FFN.tranformation_layers[0].bias
                # ))
                # print(f"Paramters loss grads were")
                # print((
                #     parameter_loss_grads["W1"],
                #     parameter_loss_grads["B1"]
                # ))

                self.FFN.update_parameters(parameter_chages)



        # foreward propagate to get cost for all test data
        costs = []
        for data_item_index in self.test_data_indexes:
            X, Y = self.X_data[data_item_index], self.Y_data[data_item_index]

            # X = X if isinstance(X, np.ndarray) else np.array([X,])
```

```
            # Y = Y if isinstance(Y, np.ndarray) else np.array([Y,])

            _, cost = self.FFN.foreward_propagate(X, Y)
            # total_cost += cost
            costs.append(cost)

        # take mean cost to be loss and state loss
        mean_cost = sum(costs) / self.num_test_data_items
        variance_cost = (
            (sum(cost**2 for cost in costs) / self.num_test_data_items)
            - mean_cost**2
        )

        return mean_cost, variance_cost

    def print_FFN_parameters(self):
        self.FFN.print_parameters()

def create_1_input_1_output_XY_data(function, num_data_items, random_x_function):
    num_data_items = 1000
    X_data = [random_x_function() for _ in range(num_data_items)]
    Y_data = [function(X_data[i]) for i in range(num_data_items)]

    X_data, Y_data = np.array(X_data), np.array(Y_data)
    return X_data, Y_data

def create_a_inputs_b_outputs_XY_data(a, b, random_x_function, function, num_data_items):
    X_data = np.array([[random_x_function() for _ in range(a)]
              for _ in range(num_data_items)
    ])

    Y_data = np.array([function(X_data[i]) for i in range(num_data_items)])

    # X_data, Y_data = np.array(X_data), np.array(Y_data)
    return X_data, Y_data

```