**Lakshya Kwatra - 2017A3PS0365P**

**Ashish Prabhune - 2017A7PS0231P**

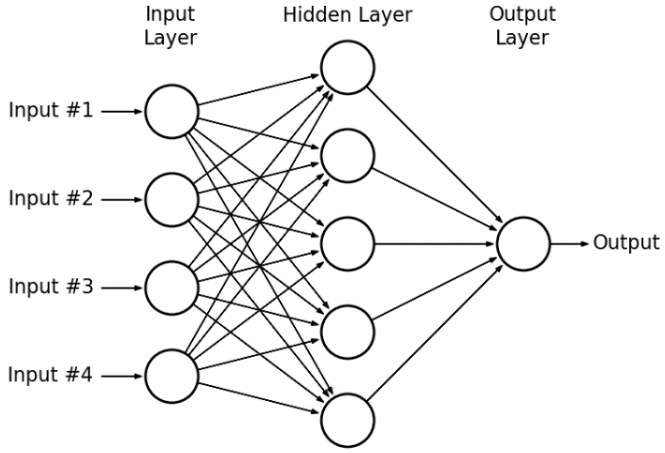
**Manasvi Agarwal – 2017A8PS0542P**

**Assignment – I**

**Machine Learning**

# Multilayer Perceptron

A **multilayer perceptron** (MLP) is a class of feedforward artificial neural network (ANN).
An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable.




## APPROACH FOLLOWED:

The approach followed by us to write the code for the MLP learning algorithm is as follows

1. The feed-forward signals from the input to the output are calculated.
2. The output error $E$ is calculated based on the predictions $a_k$ and the target $t_k$
3. The error signals are backpropagated by weighting it by the weights in previous layers and the gradients of the associated activation functions.
4. The gradients $\frac{\partial E}{\partial \theta}$ are calculated for the parameters based on the backpropagated error signal and the feedforward signals from the inputs.
5. The parameters are updated using the calculated gradients $\theta \leftarrow \theta - \eta \frac{\partial E}{\partial \theta}$
6. We have generalized the code for any number of layers.

## HELPER FUNCTIONS:

Some of the helpful functions used in the code are given as follows:

1. **Matrix Multiplication**: a (1xn), b (nxp)

```
void mat_mul(double* a, double** b, double* result, int n, int p) {
    int j, k;
    for (j = 0; j < p; j++) {
        result[j] = 0.0;
        for (k = 0; k < n; k++)
            result[j] += (a[k] * b[k][j]);
    }
}
```

2. **Randomly Shuffle:** This utility is used to randomly shuffle the data elements in an array a.

```
void randomly_shuffle(int* a, int n) {
    int i, j;
    srand(time(NULL));
    for (i = n-1; i > 0; i--) {
        j = rand() % (i+1);
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

3. **Initialize weights:** This utility is used to initialize random weights to the layer inputs of the MLP and thus, a random initialization is done between [-epsilon[i], epsilon[i]] for weight[i].

```
void initialize_weights(parameters* param, int n_layers, int* layer_sizes) {
    srand(time(0));
    double* epsilon = (double*)calloc(n_layers-1, sizeof(double));
    int i;
    for (i = 0; i < n_layers-1; i++)
        epsilon[i] = sqrt(6.0 / (layer_sizes[i] + layer_sizes[i+1])); //random
initialization
    int j, k;
    for (i = 0; i < n_layers-1; i++)
        for (j = 0; j < layer_sizes[i]+1; j++)
            for (k = 0; k < layer_sizes[i+1]; k++)
                param->weight[i][j][k] = -epsilon[i] + ((double)rand() /
((double)RAND_MAX / (2.0 * epsilon[i])));
    free(epsilon);
```

```
}
```

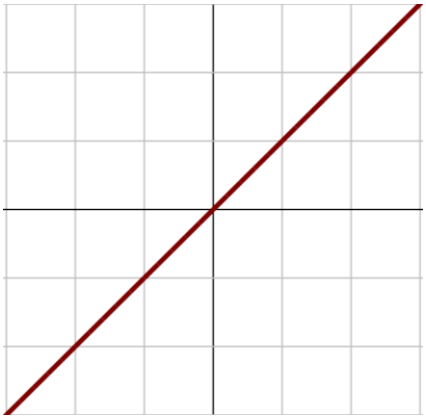
4. **Read CSV:** To read the dataset stored in csv format for data analysis.

```
void read_csv(char* filename, int rows, int cols, double** data) {
    FILE* fp = fopen(filename, "r");
    if (NULL == fp) {
        printf("Error opening %s file. Make sure you mentioned the file path
correctly\n", filename);
        exit(0);
    }
    char* line = (char*)malloc(MAX_LINE_SIZE * sizeof(char));
    int i, j;
    for (i = 0; fgets(line, MAX_LINE_SIZE, fp) && i < rows; i++) {
        char* tok = strtok(line, ",");
        for (j = 0; tok && *tok; j++) {
            data[i][j] = atof(tok);
            tok = strtok(NULL, ",\n");
        }
    }
    free(line);
    fclose(fp);
}
```

# ACTIVATION FUNCTIONS:

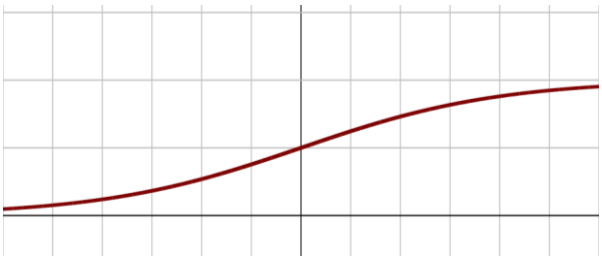
In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. Calculation of activation functions is a crucial step in the forward propagation process.

1. **Identity**: *f(x) = x*



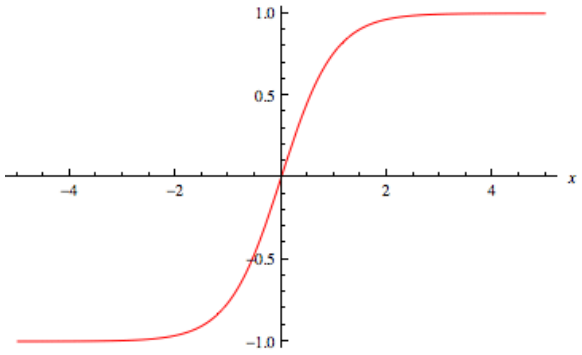
```
void identity(int n, double* input, double* output) {
output[0] = 1; //Bias
      int i;
    for (i = 0; i < n; i++)
        output[i+1] = input[i];
}
```

2. **Sigmoid**: *f(x) = 1/(1+exp(-x))*



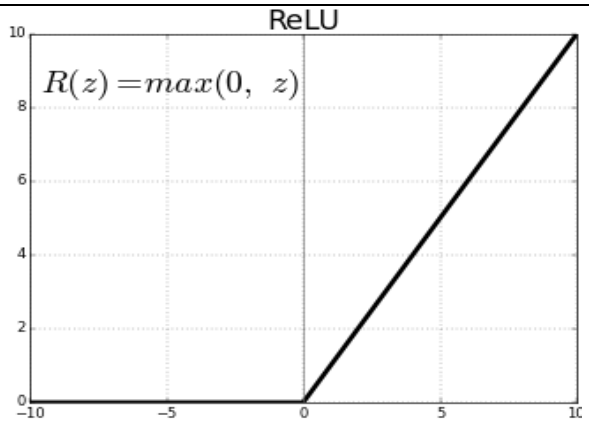
```
void sigmoid(int n, double* input, double* output) {
    output[0] = 1; //Bias
    int i;`
    for (i = 0; i < n; i++)
        output[i+1] = 1.0 / (1.0 + exp(-input[i]));
}
```

3. **Tanh**: *f(x) = tanh(x) = (exp(x) – exp(-x))/(exp(x) + exp(-x))*



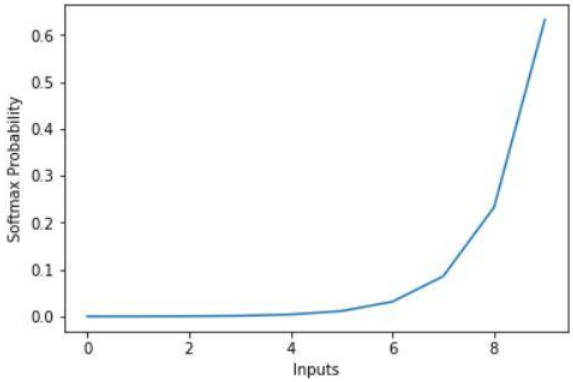

```
void tan_h(int n, double* input, double* output) {
    output[0] = 1; // Bias
    int i;
    for (i = 0; i < n; i++)
        output[i+1] = tanh(input[i]);
}
```

4. **ReLU**: $f(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$



```
void relu(int n, double* input, double* output) {
    output[0] = 1; // Bias
    int i;
    for (i = 0; i < n; i++)
        output[i+1] = max(0.0, input[i]);}
```

5. **Softmax**: $f(x_i) = exp(x_i) / \sum_{j=0}^{k} exp(x_j)$




```
void softmax(int n, double* input, double* output) {
    output[0] = 1; // Bias
    int i;
    double sum = 0.0;
    for (i = 0; i < n; i++)
        sum += exp(input[i]);
    for (i = 0; i < n; i++)
        output[i+1] = exp(input[i]) / sum;
}
```

# ACTIVATION FUNCTION DERIVATIVES:

The derivatives of different activation functions are necessary to be calculated because their values are required in the backpropagation part of the learning process so as to commit necessary changes to the input weights.

1. **Identity**: *f'(x) = 1*

```
void d_identity(int layer_size, double* layer_input, double* layer_output,
double* layer_derivative) {
    int i;
    for (i = 0; i < layer_size; i++)
        layer_derivative[i] = 1;
}
```

2. **Sigmoid**: $f'(x) = f(x)*(1 - f(x))$

```
void d_sigmoid(int layer_size, double* layer_input, double* layer_output, double*
layer_derivative) {
    int i;
    for (i = 0; i < layer_size; i++)
        layer_derivative[i] = layer_output[i+1] * (1.0 - layer_output[i+1]);
}
```

3. **Tanh**: $f'(x) = 1 - f(x)^2$

```
void d_tanh(int layer_size, double* layer_input, double* layer_output, double*
layer_derivative) {
    int i;
    for (i = 0; i < layer_size; i++)
        layer_derivative[i] = 1.0 - layer_output[i+1] * layer_output[i+1];
}
```

4. **ReLU**: $f'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$

```
void d_relu(int layer_size, double* layer_input, double* layer_output, double*
layer_derivative) {
    int i;
    for (i = 0; i < layer_size; i++) {
        if (layer_input[i] > 0)
            layer_derivative[i] = 1;
        else if (layer_input[i] < 0)
            layer_derivative[i] = 0;
        else
            layer_derivative[i] = 0.5;
    }
}
```

5. **Softmax**: $f(x_i) = f(x)*(1 - f(x))$

```
void d_softmax(int layer_size, double* layer_input, double* layer_output, double*
layer_derivative) {
    int i;
    for (i = 0; i < layer_size; i++)
        layer_derivative[i] = layer_output[i+1] * (1.0 - layer_output[i+1]);
}
```

# LAYERS:

The MLP consists of three or more layers (an input and an output layer with one or more *hidden layers*) of nonlinearly-activating nodes. Since MLPs are fully connected, each node in one layer connects with a certain weight $w_{ij}$ to every node in the following layer.

1. **Forward Propagation:**

As the name suggests, the input data is fed in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function and passes to the successive layer. Given below is the function definition of forward_propagation.

```
void forward_propagation(parameters* param, int training_example, int n_layers,
int* layer_sizes, double** layer_inputs, double** layer_outputs) {
```

Firstly, we would find the input layer's input and output (both are equal) from data matrix with the given training example and then, we'll perform forward propagation for every hidden layer.

```
    int i;
    layer_outputs[0][0] = 1; // Bias term of input layer
    for (i = 0; i < param->feature_size-1; i++)
        layer_outputs[0][i+1] = layer_inputs[0][i] = param-
>data_train[training_example][i];
    // Calculating input and output of each hidden layer
    for (i = 1; i < n_layers-1; i++) {
        mat_mul(layer_outputs[i-1], param->weight[i-1], layer_inputs[i],
layer_sizes[i-1]+1, layer_sizes[i]);
        if (param->hidden_activation_functions[i-1] == 1)
           identity(layer_sizes[i], layer_inputs[i], layer_outputs[i]);
             else if (param->hidden__activation_functions[i-1] == 2)
                sigmoid(layer_sizes[i], layer_inputs[i], layer_outputs[i]);
        else if (param->hidden__activation_functions[i-1] == 3)
                tan_h(layer_sizes[i], layer_inputs[i], layer_outputs[i]);
        else if (param->hidden__activation_functions[i-1] == 4)
                relu(layer_sizes[i], layer_inputs[i], layer_outputs[i]);
        else if (param->hidden__activation_functions[i-1] == 5)
                softmax(layer_sizes[i], layer_inputs[i], layer_outputs[i]);
        else{
                printf("Invalid hidden activation function\n");
                exit(0);}
    }
```

Now we will be performing forward propagation for every hidden layer.

```
    mat_mul(layer_outputs[n_layers-2], param->weight[n_layers-2],
layer_inputs[n_layers-1], layer_sizes[n_layers-2]+1, layer_sizes[n_layers-1]);
```

```
    if (param->output_activation_function == 1)
            identity(layer_sizes[n_layers-1], layer_inputs[n_layers-1],
layer_outputs[n_layers-1]);
    else if (param->output_activation_function == 2)
            sigmoid(layer_sizes[n_layers-1], layer_inputs[n_layers-1],
layer_outputs[n_layers-1]);
    else if (param->output_activation_function == 3)
                tan_h(layer_sizes[n_layers-1], layer_inputs[n_layers-1],
layer_outputs[n_layers-1]);
    else if (param->output_activation_function == 4)
            relu(layer_sizes[n_layers-1], layer_inputs[n_layers-1],
layer_outputs[n_layers-1]);
    else if (param->output_activation_function == 5)
            softmax(layer_sizes[n_layers-1], layer_inputs[n_layers-1],
layer_outputs[n_layers-1]);
    else{
           printf("Forward propagation: Invalid hidden activation function\n");
                exit(0);}

}
```

# LEARNING:

Learning occurs in the perceptron by changing connection weights after each piece of data is processed, based on the amount of error in the output compared to the expected result. This is an example of supervised learning, and is carried out through backpropagation, a generalization of the least mean squares algorithm in the linear perceptron.

1. **Output Error and Local Gradient:**

In this step, the output error is calculated based on the predictions and the target. Following which, the error signals are backpropagated by weighting it by the weights in previous layers and the gradients of the associated activation functions. Given below is the function definition for calculate_local_gradient.
We start by creating some memory for the layer derivatives using calloc().

```
void calculate_local_gradient(parameters* param, int layer_no, int n_layers, int*
layer_sizes, double** layer_inputs, double** layer_outputs,
    double* expected_output, double** local_gradient) {
    double** layer_derivatives = (double**)calloc(n_layers, sizeof(double*));
    int i;
    for (i = 0; i < n_layers; i++)
        layer_derivatives[i] = (double*)calloc(layer_sizes[i], sizeof(double));
```

If the layer we are dealing with is the output layer, then we will calculate the error produced at the output and consequently, we will find the corresponding local gradient.

```
if (layer_no == n_layers-1) {
        // Error produced at the output layer
        double* error_output = (double*)calloc(param->output_layer_size,
sizeof(double));
        for (i = 0; i < param->output_layer_size; i++)
           error_output[i] = expected_output[i] - layer_outputs[layer_no][i+1];
        // Calculate the layer derivatives and local gradients
        if(param->output_activation_function == 1){
           d_identity(param->output_layer_size, layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
           for (i = 0; i < param->output_layer_size; i++)
                local_gradient[layer_no][i] = error_output[i] *
layer_derivatives[layer_no][i];
         }
        else if(param->output_activation_function == 2){
           d_sigmoid(param->output_layer_size, layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
           for (i = 0; i < param->output_layer_size; i++)
                local_gradient[layer_no][i] = error_output[i] *
layer_derivatives[layer_no][i];
          }
        else if(param->output_activation_function == 3){
           d_tanh(param->output_layer_size, layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
           for (i = 0; i < param->output_layer_size; i++)
                local_gradient[layer_no][i] = error_output[i] *
layer_derivatives[layer_no][i];
          }
        else if(param->output_activation_function == 4){
           d_relu(param->output_layer_size, layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
           for (i = 0; i < param->output_layer_size; i++)
                local_gradient[layer_no][i] = error_output[i] *
layer_derivatives[layer_no][i];
          }
else if(param->output_activation_function == 5){
           d_softmax(param->output_layer_size, layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
           for (i = 0; i < param->output_layer_size; i++)
                local_gradient[layer_no][i] = error_output[i] *
layer_derivatives[layer_no][i];
```

```
        }
else{
     printf("Calculate local gradient: Invalid output activation function\n");
     exit(0);
    }
free(error_output);
}
```

If the layer we are dealing with is a hidden layer, then we calculate the layer derivative for all units in the layer and consequently, we calculate the local gradient.

```
else {
        int j;
        if(param->hidden_activation_functions[layer_no-1] == 1) {
                d_identity(layer_sizes[layer_no], layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
                for (i = 0; i < layer_sizes[layer_no]; i++) {
                    double error = 0.0;
                    for (j = 0; j < layer_sizes[layer_no+1]; j++)
                        error += local_gradient[layer_no+1][j] * param-
>weight[layer_no][i][j];
                    local_gradient[layer_no][i] = error *
layer_derivatives[layer_no][i];
                }
           }
        else if(param->hidden_activation_functions[layer_no-1] == 2) {
                d_sigmoid(layer_sizes[layer_no], layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
                for (i = 0; i < layer_sizes[layer_no]; i++) {
                    double error = 0.0;
                    for (j = 0; j < layer_sizes[layer_no+1]; j++)
                        error += local_gradient[layer_no+1][j] * param-
>weight[layer_no][i][j];
                    local_gradient[layer_no][i] = error *
layer_derivatives[layer_no][i];
                }
           }
        else if(param->hidden_activation_functions[layer_no-1] == 3) {
                d_tanh(layer_sizes[layer_no], layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
                for (i = 0; i < layer_sizes[layer_no]; i++) {
                    double error = 0.0;
                    for (j = 0; j < layer_sizes[layer_no+1]; j++)
```

```
                        error += local_gradient[layer_no+1][j] * param-
>weight[layer_no][i][j];
                    local_gradient[layer_no][i] = error *
layer_derivatives[layer_no][i];
                }
            }

        else if(param->hidden_activation_functions[layer_no-1] == 4) {
                d_relu(layer_sizes[layer_no], layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
                for (i = 0; i < layer_sizes[layer_no]; i++) {
                    double error = 0.0;
                    for (j = 0; j < layer_sizes[layer_no+1]; j++)
                        error += local_gradient[layer_no+1][j] * param-
>weight[layer_no][i][j];
                    local_gradient[layer_no][i] = error *
layer_derivatives[layer_no][i];
                }
            }
       else if(param->hidden_activation_functions[layer_no-1] == 5) {
                d_softmax(layer_sizes[layer_no], layer_inputs[layer_no],
layer_outputs[layer_no], layer_derivatives[layer_no]);
                for (i = 0; i < layer_sizes[layer_no]; i++) {
                    double error = 0.0;
                    for (j = 0; j < layer_sizes[layer_no+1]; j++)
                        error += local_gradient[layer_no+1][j] * param-
>weight[layer_no][i][j];
                    local_gradient[layer_no][i] = error *
layer_derivatives[layer_no][i];
                }
            }
       else{
            printf("Invalid hidden activation function\n");
            exit(0);
            }
     }
    for (i = 0; i < n_layers; i++)
        free(layer_derivatives[i]);
    free(layer_derivatives);
}
```

Now that we have produced a utility for the calculation of error outputs and thus the local gradient descents, we can use it in the backpropagation step of the learning process of our ANN. Given below is the function definition of back_propagation.

```
void back_propagation(parameters* param, int training_example, int n_layers, int*
layer_sizes, double** layer_inputs, double** layer_outputs) {
```

Now, we would get the expected output from the data matrix. We first create memory for the expected output array with all elements initialized to zeros. We would make the respective element in expected_output to 1 and rest all 0.

```
  double* expected_output = (double*)calloc(param->output_layer_size,
sizeof(double));
    if (param->output_layer_size == 1)
        expected_output[0] = param->data_train[training_example][param-
>feature_size-1];
    else
        expected_output[(int)(param->data_train[training_example][param-
>feature_size-1] - 1)] = 1;
```

After this, we need to allocate some memory for the weight correction matrices between the layers, where weight_correction is a pointer to the array of 2D arrays between the layers and each 2D array between two layers i and i+1 is of size ((layer_size[i]+1) x layer_size[i+1]). The weight_correction matrix includes weight corrections for the bias terms as well.

```
    double*** weight_correction = (double***)calloc(n_layers-1,
sizeof(double**));
    int i;
    for (i = 0; i < n_layers-1; i++)
        weight_correction[i] = (double**)calloc(layer_sizes[i]+1,
sizeof(double*));
    int j;
    for (i = 0; i < n_layers-1; i++)
        for (j = 0; j < layer_sizes[i]+1; j++)
            weight_correction[i][j] = (double*)calloc(layer_sizes[i+1],
sizeof(double));
```

Now we allocate some memory for the local gradient (delta) of each layer.

```
    double** local_gradient = (double**)calloc(n_layers, sizeof(double*));
    for (i = 0; i < n_layers; i++)
        local_gradient[i] = (double*)calloc(layer_sizes[i], sizeof(double));
```

Now, we calculate the weight corrections for all layers' weights (output and hidden layers) by making use of the local_gradient_utility that we've produced earlier..

```
 calculate_local_gradient(param, n_layers-1, n_layers, layer_sizes,
layer_inputs, layer_outputs, expected_output, local_gradient);
    for (i = 0; i < param->output_layer_size; i++)
        for (j = 0; j < layer_sizes[n_layers-2]+1; j++)
            weight_correction[n_layers-2][j][i] = (param->learning_rate) *
local_gradient[n_layers-1][i] * layer_outputs[n_layers-2][j];
    int k;
    for (i = n_layers-2; i >= 1; i--) {
        calculate_local_gradient(param, i, n_layers, layer_sizes, layer_inputs,
layer_outputs, expected_output, local_gradient);

        for (j = 0; j < layer_sizes[i]; j++)
            for (k = 0; k < layer_sizes[i-1]+1; k++)
                weight_correction[i-1][k][j] = (param->learning_rate) *
local_gradient[i][j] * layer_outputs[i-1][k];
    }
```

Finally, we update the weights, and free the heap allocated memory.

```
    for (i = 0; i < n_layers-1; i++) {
        for (j = 0; j < layer_sizes[i]+1; j++) {
            for (k = 0; k < layer_sizes[i+1]; k++) {
                param->weight[i][j][k] -= weight_correction[i][j][k];
            }
        }
    }
    for (i = 0; i < n_layers; i++)
        free(local_gradient[i]);
    free(local_gradient);
    for (i = 0; i < n_layers - 1; i++)
        for (j = 0; j < layer_sizes[i]+1; j++)
            free(weight_correction[i][j]);
    for (i = 0; i < n_layers - 1; i++)
        free(weight_correction[i]);
    free(weight_correction);
    free(expected_output);
}
```

# TRAINING:

Now that we have all the function required for the forward propagation, backpropagation and weight updates, we can incorporate all these in a single function for the ease of use, Given below is the function definition of the utility mlp_trainer to achieve the same.

```
void mlp_trainer(parameters* param, int* layer_sizes) {
    // Total number of layers
    int n_layers = param->n_hidden + 2;

    double** layer_inputs = (double**)calloc(n_layers, sizeof(double*));
    int i;
    for (i = 0; i < n_layers; i++)
        layer_inputs[i] = (double*)calloc(layer_sizes[i], sizeof(double));
    double** layer_outputs = (double**)calloc(n_layers, sizeof(double*));
    for (i = 0; i < n_layers; i++)
        layer_outputs[i] = (double*)calloc(layer_sizes[i]+1, sizeof(double));
    initialize_weights(param, n_layers, layer_sizes);
    int* indices = (int*)calloc(param->train_sample_size, sizeof(int));
    for (i = 0; i < param->train_sample_size; i++)
        indices[i] = i;
    int training_example, j;
    for (i = 0; i < param->n_iterations_max; i++) {
        printf("Iteration %d of %d(max)\r", i+1, param->n_iterations_max);
        randomly_shuffle(indices, param->train_sample_size);
        for (j = 0; j < param->train_sample_size; j++) {
            training_example = indices[j];
            forward_propagation(param, training_example, n_layers, layer_sizes,
layer_inputs, layer_outputs);
            back_propagation(param, training_example, n_layers, layer_sizes,
layer_inputs, layer_outputs);
        }
    }
    free(indices);
    for (i = 0; i < n_layers; i++)
        free(layer_outputs[i]);

    free(layer_outputs);

    for (i = 0; i < n_layers; i++)
        free(layer_inputs[i]);
    free(layer_inputs);
}
```

# DATASET EXPLANATION:

1. **Binary Classification – Bank Note Authentication**
   Data was extracted from images that were taken from genuine and forged banknote-like specimens. For digitization, an industrial camera usually used for print inspection was used. The final images have 400x

400 pixels. Due to the object lens and distance to the investigated object gray-scale pictures with a resolution of about 660 dpi were gained. Wavelet Transform tool were used to extract features from images. The attributes taken are:

1. Variance of wavelet transformed image
2. Skewness of wavelet transformed image
3. Entropy of image
4. Class

The following table gives the statistical description of various attributes of the dataset:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| count | 1371.000000 | 1371.000000 | 1371.000000 | 1371.000000 | 1371.000000 |
| mean | 0.431410 | 1.917434 | 1.400694 | -1.192200 | 0.444931 |
| std | 2.842494 | 5.868359 | 4.310105 | 2.101683 | 0.497139 |
| min | -7.042100 | -13.773100 | -5.286100 | -8.548200 | 0.000000 |
| 25% | -1.774700 | -1.711300 | -1.553350 | -2.417000 | 0.000000 |
| 50% | 0.495710 | 2.313400 | 0.616630 | -0.586650 | 0.000000 |
| 75% | 2.814650 | 6.813100 | 3.181600 | 0.394810 | 1.000000 |
| max | 6.824800 | 12.951600 | 17.927400 | 2.449500 | 1.000000 |

**2. Multi-Class Classification: Wine Quality Dataset**

The dataset is related to red variant of the Portuguese "Vinho Verde" wine. The classes are ordered and not balanced (e.g. there are many more normal wines than excellent or poor ones). Outlier detection algorithms could be used to detect the few excellent or poor wines. Also, all input variables are relevant. The attributes taken are:

1. Fixed acidity
2. Citric acid
3. Chlorides
4. Free sulfur oxide
5. Density
6. pH
7. Sulphates
8. Alcohol
9. Quality (between 0 and 10)

The following table gives the statistical description of various attributes of the dataset:

| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 1598.000000 | 1598.000000 | 1598.000000 | 1598.000000 | 1598.000000 | 1598.000000 | 1598.000000 | 1598.000000 | 1598.000000 | 1598.000000 | 1598.000000 |
| mean | 8.320213 | 0.527713 | 0.271145 | 2.539205 | 0.087474 | 15.877972 | 46.475594 | 0.996746 | 3.310989 | 0.658210 | 10.423623 |
| std | 1.741489 | 0.179064 | 0.194744 | 1.410279 | 0.047079 | 10.462720 | 32.904142 | 0.001888 | 0.154355 | 0.169542 | 1.065694 |
| min | 4.600000 | 0.120000 | 0.000000 | 0.900000 | 0.012000 | 1.000000 | 6.000000 | 0.990070 | 2.740000 | 0.330000 | 8.400000 |
| 25% | 7.100000 | 0.390000 | 0.090000 | 1.900000 | 0.070000 | 7.000000 | 22.000000 | 0.995600 | 3.210000 | 0.550000 | 9.500000 |
| 50% | 7.900000 | 0.520000 | 0.260000 | 2.200000 | 0.079000 | 14.000000 | 38.000000 | 0.996750 | 3.310000 | 0.620000 | 10.200000 |
| 75% | 9.200000 | 0.640000 | 0.420000 | 2.600000 | 0.090000 | 21.000000 | 62.000000 | 0.997837 | 3.400000 | 0.730000 | 11.100000 |
| max | 15.900000 | 1.580000 | 1.000000 | 15.500000 | 0.611000 | 72.000000 | 289.000000 | 1.003690 | 4.010000 | 2.000000 | 14.900000 |

# RESULTS:

1. **Binary Classification:**

The classification has been done on the banknote authentication UCI ML dataset. The best accuracy achieved by the MLP with one hidden layer is 78.91% with 4 Neurons in the hidden layer and Identity as the activation Function.

| Activation Function | No. of Neurons in hidden Layer | No. of Hidden Layers | Accuracy |
|---|---|---|---|
| Identity | 2 | 1 | 72.73% |
| Identity | 3 | 1 | 78.55% |
| Identity | 4 | 1 | 78.91% |
| Sigmoid | 2 | 1 | 61.82% |
| Sigmoid | 3 | 1 | 58.18% |
| Sigmoid | 4 | 1 | 68.36% |
| ReLU | 2 | 1 | 63.64% |
| ReLU | 3 | 1 | 70.55% |
| ReLU | 4 | 1 | 72.00% |
| Tanh | 2 | 1 | 72.00% |
| Tanh | 3 | 1 | 62.55% |
| Tanh | 4 | 1 | 52% |
| Softmax | 2 | 1 | 54.18% |
| Softmax | 3 | 1 | 55.20% |
| Softmax | 4 | 1 | 50.91% |

Given Below are some of the Confusion Matrices related to the above learning processes:

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 57 | 106 |
| ACTUAL 1 | 29 | 83 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 80 | 83 |
| ACTUAL 1 | 43 | 69 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 32 | 131 |
| ACTUAL 1 | 1 | 111 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 115 | 48 |
| ACTUAL 1 | 55 | 57 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 152 | 11 |
| ACTUAL 1 | 66 | 46 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 157 | 6 |
| ACTUAL 1 | 71 | 41 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 159 | 4 |
| ACTUAL 1 | 77 | 35 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 163 | 0 |
| ACTUAL 1 | 100 | 12 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 116 | 47 |
| ACTUAL 1 | 40 | 72 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 155 | 8 |
| ACTUAL 1 | 97 | 15 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 158 | 5 |
| ACTUAL 1 | 53 | 59 |

| | PREDICTED 0 | PREDICTED 1 |
|---|---|---|
| ACTUAL 0 | 162 | 1 |
| ACTUAL 1 | 58 | 54 |

2. **Multiclass Classification:**

The classification has been done on the red-wine quality UCI ML dataset. The best accuracy achieved by the MLP with one hidden layer is 63.66% with 4 Neurons in the hidden layer and Sigmoid as the activation Function.

| Activation Function | No. of Neurons in hidden Layer | No. of Hidden Layers | Accuracy |
|---|---|---|---|
| Identity | 2 | 1 | 37.27% |
| Identity | 3 | 1 | 44.76% |
| Identity | 4 | 1 | 60.08% |
| Sigmoid | 2 | 1 | 22.65% |
| Sigmoid | 3 | 1 | 53.63% |
| Sigmoid | 4 | 1 | 63.66% |
| ReLU | 2 | 1 | 23.49% |

| | | | |
|---|---|---|---|
| ReLU | 3 | 1 | 54.16% |
| ReLU | 4 | 1 | 41.22% |
| Tanh | 2 | 1 | 39.30% |
| Tanh | 3 | 1 | 33.34% |
| Tanh | 4 | 1 | 46.00% |
| Softmax | 2 | 1 | 24.72% |
| Softmax | 3 | 1 | 35.03% |
| Softmax | 4 | 1 | 41.16% |

## CONCLUSIONS:

1. For the chosen datasets, in case of binary classification, the model seems to work the best for identity activation function and in case of multiclass classification, the model seems to work the best for Sigmoid activation function
2. Having tried Batch, Stochastic and Minibatch gradient descents in our data, the best results came when we used the Minibatch Gradient descent