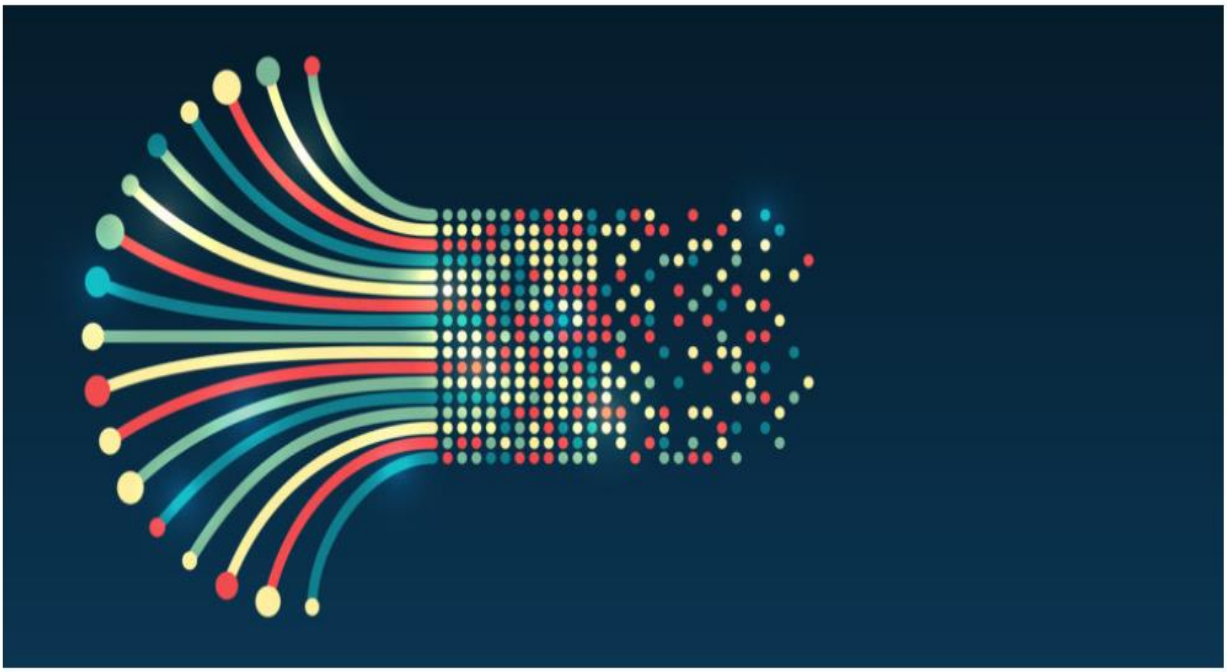
Pattern Recognition

Assignment 1 – Report

Implementation of Backpropagation and Neural Network



Students:

Rana Mohamed Zayed	7756
Saifallah Mohamed Seddik	7696
Mazen Gaber Ibrahim	7467

1. Introduction:

The project emphasizes the significance of various components of neural networks, such as gates, activation functions, and loss functions. By implementing these components from scratch, as well as developing a comprehensive model of the implemented gates with applying enhancement optimization factors for better execution.

2. Main Implementation:

2.1 Gates, Activation and Loss functions:

First, the basic unit of the neural network is built from scratch by implementing fundamental gates such as: addition, multiplication, division, subtraction, power,, exponential and more trigonometric functions such as: sine, cosine and tan. Each of these functions are implemented in both their forward and backward propagation.

Moreover, the activation functions were implemented in the same way containing the ReLU, Linear, Sigmoid, Softmax and other necessary functions.

Then, loss functions are also provided such as binary cross-entropy and L2 loss.

2.2 Implementing Computational Graph / Model

Model architecture:

The neural network model is implemented as a Python class named Model. It consists of the following key components:

Initialization:

The '__init__' method initializes the model with parameters such as 'layers_dim', 'activation_func', and 'loss'. These parameters define the architecture of the neural network, including the number of layers, activation functions for each layer, and the loss function to be used during training.

Parameter Initialization:

The initialize_parameters method initializes the weights and biases of the neural network. Depending on the activation function specified for each layer, appropriate weight initialization techniques such as Xavier/Glorot initialization or He initialization are applied to ensure proper initialization for better convergence and training stability.

Forward Propagation:

The forward_propagation method computes the forward pass of the neural network. It takes input data X and propagates it through the network, applying activation functions at each layer to compute the output of the network.

Backward Propagation:

The backward_propagation method computes the backward pass of the neural network, which involves calculating gradients of the loss function with respect to the parameters (weights and biases) of the network. This step enables updating the parameters of the network to minimize the loss during training.

Parameter Update:

The update_parameters method updates the parameters (weights and biases) of the network using the gradients computed during backward propagation. It applies gradient descent optimization to update the parameters in the direction that minimizes the loss.

Training:

The train method trains the neural network using the specified training data and parameters such as learning rate, number of epochs, and gradient descent method. It iterates over the training data for multiple epochs, computing forward and backward passes at each iteration, and updating the parameters to minimize the loss.

Prediction:

The predict method enables making predictions using the trained neural network. Given input data X, it performs forward propagation to compute the output of the network, which represents the predicted values.

Key Features:

- **Customizable architecture:** The model allows flexibility in defining the number of layers, activation functions, and loss functions, enabling customization according to the requirements of the machine learning task.
- **Efficient training:** The model implements efficient forward and backward propagation algorithms, facilitating the training process and enabling convergence to optimal parameter values.
- **Visualization:** The model provides visualization of the training process by plotting the loss curve over epochs, allowing users to monitor the training progress and performance.

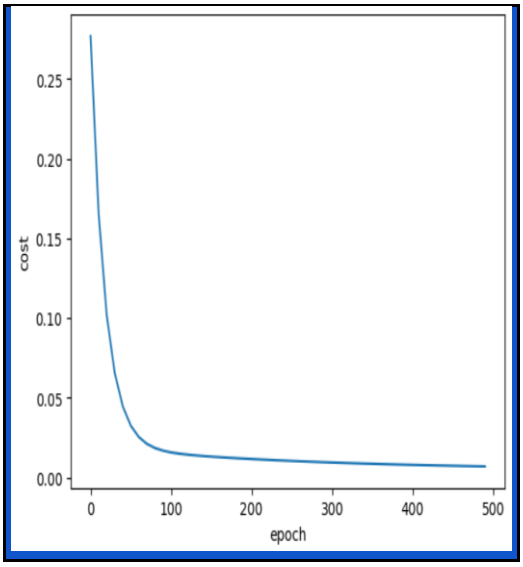
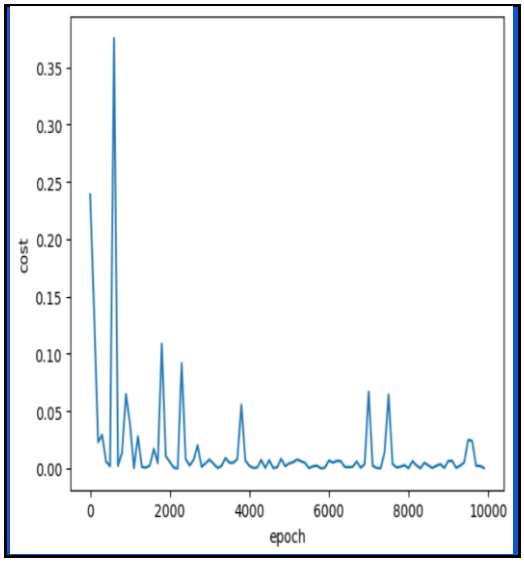
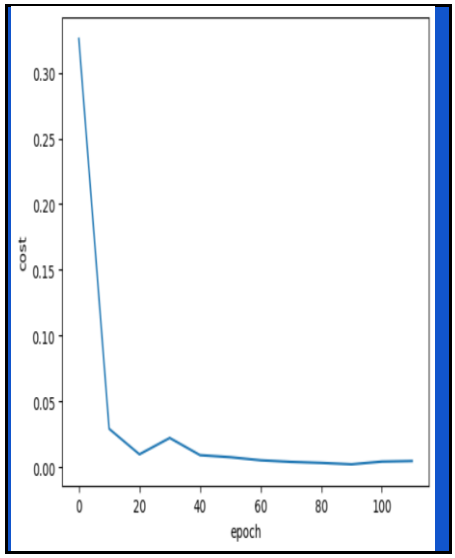
- **2.3 Implementing Gradient Descent Algorithms:**

First, Batch Gradient Descent is implemented with scalar weight to explain the Idea of gradient descent in general and compare the cost w.r.t the iterations before using G.D and after the optimization by plotting a cost graph.

Secondly, each type from the following gradients : Batch, Stochastic, Mini-Batch is implemented from scratch using *vectorized implementation.*

To get more familiar with the difference between gradients a plot for each type is provided.

Sample Runs of the cost plots through epochs:

Batch G.D	S.G.D	Mini-Batch G.D
		

Batch Gradient Descent	Stochastic Gradient Descent (SGD)	Mini-Batch Gradient Descent
• Entire dataset for updation • Cost function reduces smoothly • Computation cost is very high	• Single observation for updation • Lot of variations in cost function • Computation time is more	• Subset of data for updation • Smoother cost function as compared to SGD • Computation time is lesser than SGD • Computation cost is lesser than Batch Gradient Descent

2.4 Regularization Techniques:

- **L1-Regularization(Lasso):**

  This class encapsulates the functionality required to perform Lasso Regression, including initialization, fitting the model, updating weights through gradient descent, and making predictions. It provides a structured and modular approach to implementing Lasso Regression-L1 Regularization, making it easier to use and maintain.

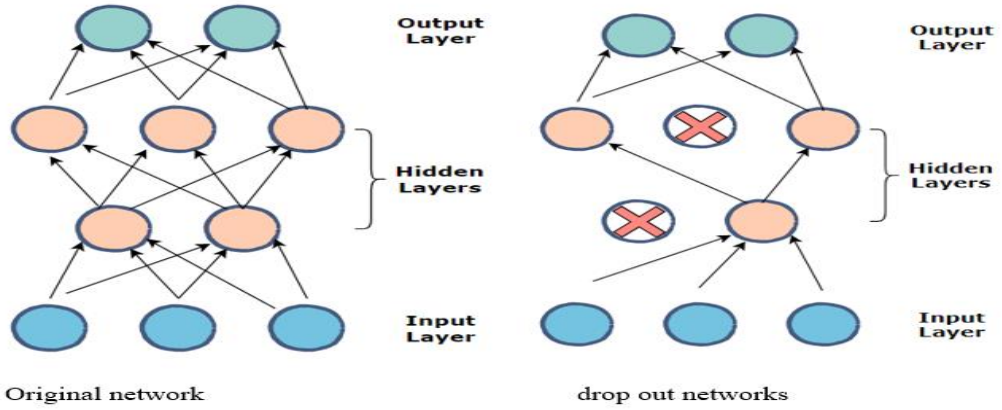
L1 Regularization

$$\text{Modified loss function} = \text{Loss function} + \lambda \sum_{i=1}^{n} |W_i|$$

- **Dropout Layer:**

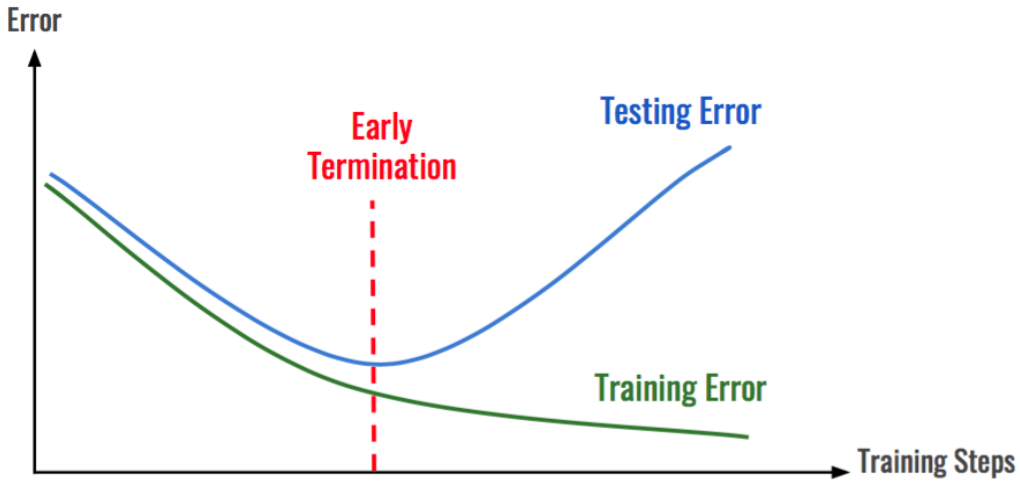
  Overall, this Dropout class encapsulates the functionality required for applying dropout regularization during both the forward and backward passes of a neural network. During training, it randomly zeros out some neurons to prevent overfitting, while

during testing, it passes the input through unchanged for accurate inference.



- **Early Stopping:**

This class encapsulates the functionality required for early stopping during model training, allowing the training process to terminate efficiently when validation performance no longer improves. It provides flexibility through parameters such as patience, min_delta, and restore_best_weights, enabling customization based on specific training requirements.



3. Bonus part Implementation:

3.1 Learning Rate Scheduler:

During different epochs, the learning rate is changed with different techniques for multiple reasons such as avoiding the model overfitting mainly. The three techniques implemented are as follows:

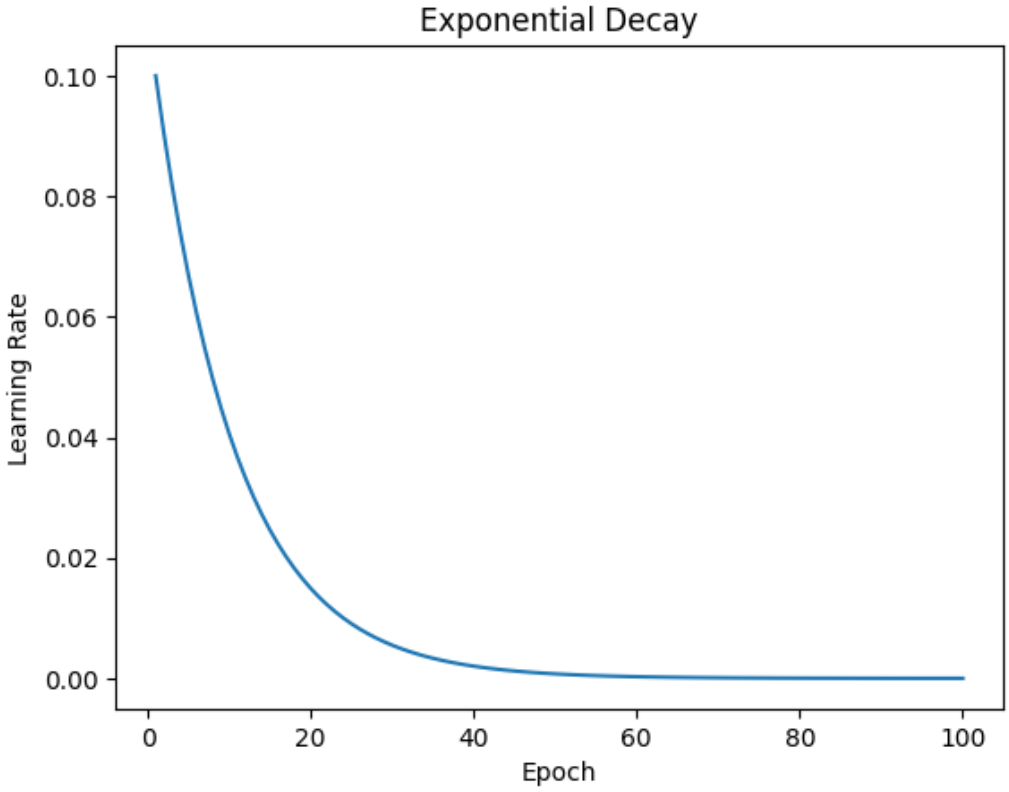
1- Exponential Decay:

Using the formula given by Stanford University, the learning rate was updated among the epochs with a decay as follows:

exponential decay:

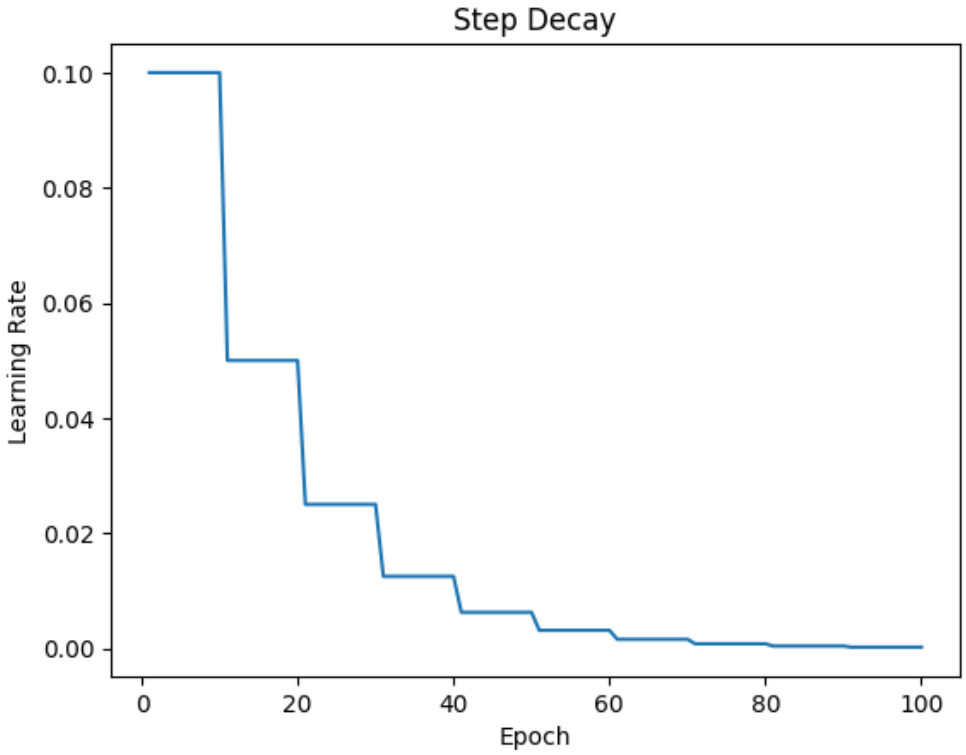
$$\alpha = \alpha_0 e^{-kt}$$

Decay curve was resulted as follows:



2- Step Decay:

Each few epochs, the learning rate was decreased by a certain step value giving the curve as follows:



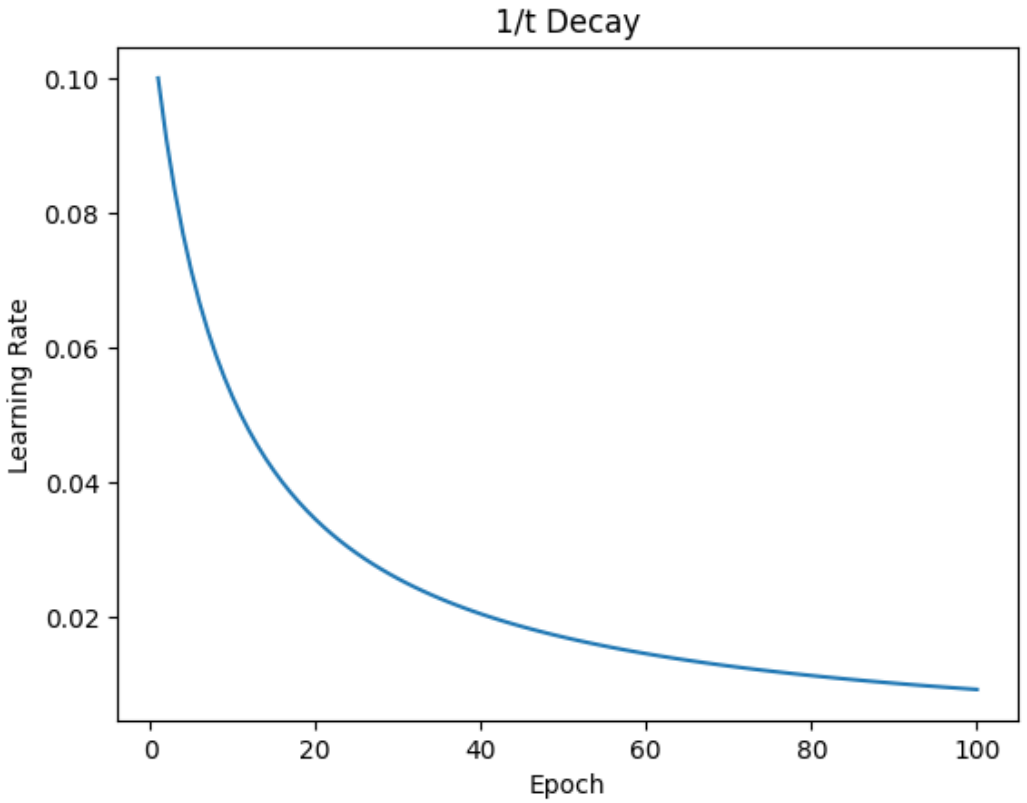
2- 1/t Decay:

Using the formula given by Stanford University, the learning rate was updated among the epochs with a decay as follows:

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Decay curve was resulted as follows:



4. Results and Analysis:

3.2 Hyperparameter Tuning:

Grid Search Algorithm:

The grid_search method in the HyperparameterTuner class implements the grid search algorithm. It systematically explores all possible combinations of hyperparameters defined in the hyperparameters_grid. For each combination, it creates an instance of the neural network model with the specified hyperparameters and trains it using the training data. After training, the model's performance is evaluated using the validation data, and the validation loss is computed. The algorithm keeps track of the model with the lowest validation loss and returns it as the best model.

Key Features:

Customizable Hyperparameters: The hyperparameter tuning algorithm allows customization of hyperparameters such as learning rate, batch size, and number of layers. Users can define the grid of hyperparameters to be explored based on their specific requirements.

Efficient Exploration: Grid search systematically explores all possible combinations of hyperparameters, ensuring thorough exploration of the hyperparameter space. This enables the identification of the optimal combination that yields the best performance.

Performance Evaluation: The algorithm evaluates each model's performance using a validation dataset, enabling unbiased assessment of model performance and preventing overfitting.

3.3 Batch Normalization:

```
Batchnorm_forward
```

This method enables batch normalization in neural networks, ensuring stable and efficient training by normalizing the input data and maintaining running statistics during training and testing phases. It provides flexibility in terms of mode selection and supports customization through optional parameters like epsilon and momentum.

```
Batchnorm_backward_alt
```

This function efficiently computes the gradients required for backpropagation in a batch normalization layer, facilitating stable and effective training of neural networks.