**A Gradient Free Neural Network Framework Based**

**on Universal Approximation Theorem**

**A Python Implementation**

The main focus of this report is to provide an understanding of the paper *A Gradient Free Neural Network Framework Based on Universal Approximation Theorem*, by Dr. Bakas, Langousis, Nicolaou & Chatzichristofis, and implement the concluding algorithm using Python. The report focuses on the mathematical explanation of the implementation.

The academic paper introduces a numerical scheme for calculating ANN weights directly, bypassing the traditional iterative optimization processes & addressing key challenges such as overfitting or computational inefficiency.

By leveraging local neighborhoods (to ensure matrix invertibility), the method derives the number of neurons dynamically from the dimensionality of the data. As stated in the paper, this approach simplifies ANN training & achieves higher accuracy for regression, classification & solving PDEs, with low errors.

The proposed algorithm has polynomial time complexity, contrasting sharply with the NP-complete nature of conventional training methods & demonstrates strong generalization capabilities.

**STUDENT:** Nikolaos Tassopoulos (272062) **INSTRUCTOR:** Dr. I. Christou

**ASSIGNMENT:** Work Study Report

**TERM:** Spring Semester 2025 **DATE:** Thursday 3rd, April 2025

**INTRODUCTION**

This paper is about implementing a neural network which combines Sigmoid-Based training & Radial Basis Function (RBF-Based) training, targeted for the loading & processing of high-dimensional data, from databases like MNIST.

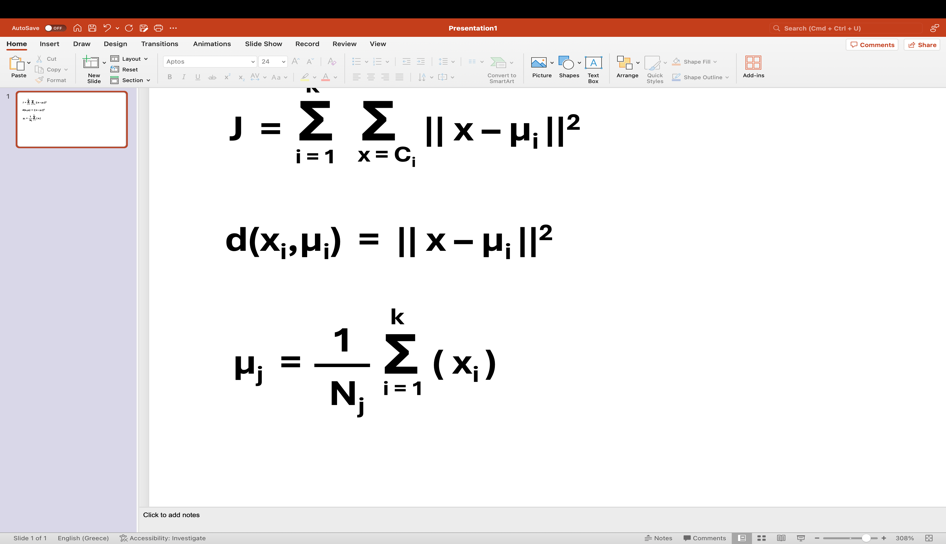
The main idea is to first apply **Clustering** to the dataset, using K-Means, meaning that the dataset is divided, with each of the clusters representing a neuron of the first layer. After that, the data are processed through **RBF & Sigmoid transformations**, so we can both introduce nonlinearity & enable the model to capture more complex patterns in the dataset (through Sigmoid) and also enhance approximation for high-dimensional datasets (through RBF). Finally, as the model follows a gradient-free approach, the paper applies the **Universal Approximation Theorem**, meaning that the final output weights are directly computed from the processed data.

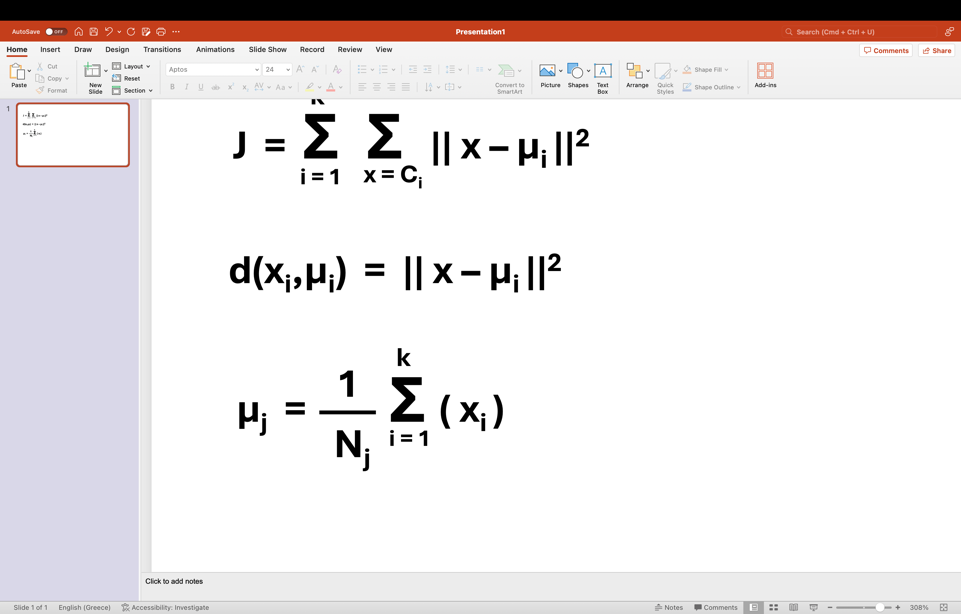
**THE CLUSTERING ALGORITHM**

The model uses K-Means Clustering, in order to organize the dataset into clusters, before the training. The goal is to ensure that each neuron will be trained with a well-defined sample of data

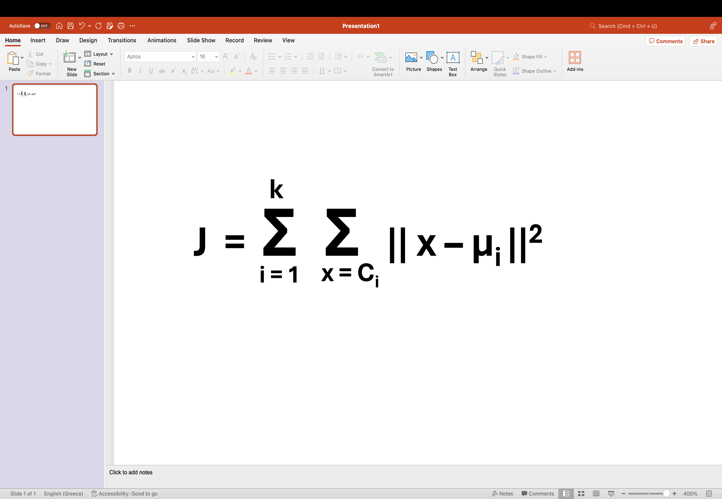
In this one, the dataset samples are balanced by extracting the same number of samples from each category of digit, from 0 to 9. After that, the k-means function is applied. The k-means algorithm initializes the number of clusters (***n\_clusters***), equalizing them with the number of neurons that we have set for the model.

The algorithm randomly selects ***k*** initial centroids & each data subset is assigned to the nearest one, using the Euclidean Distance from mathematics. The centroids are calculated by averaging the data subsets of each cluster & this process repeats until a maximum number of iterations is reached.

The data assignment process, where for each data point ***xi***, the closest centroid ***μi*** is located. It is basically the mathematical expression for Euclidean Distance!

The new centroids calculation, by using the mean for all data points that are assigned to the corresponding cluster.

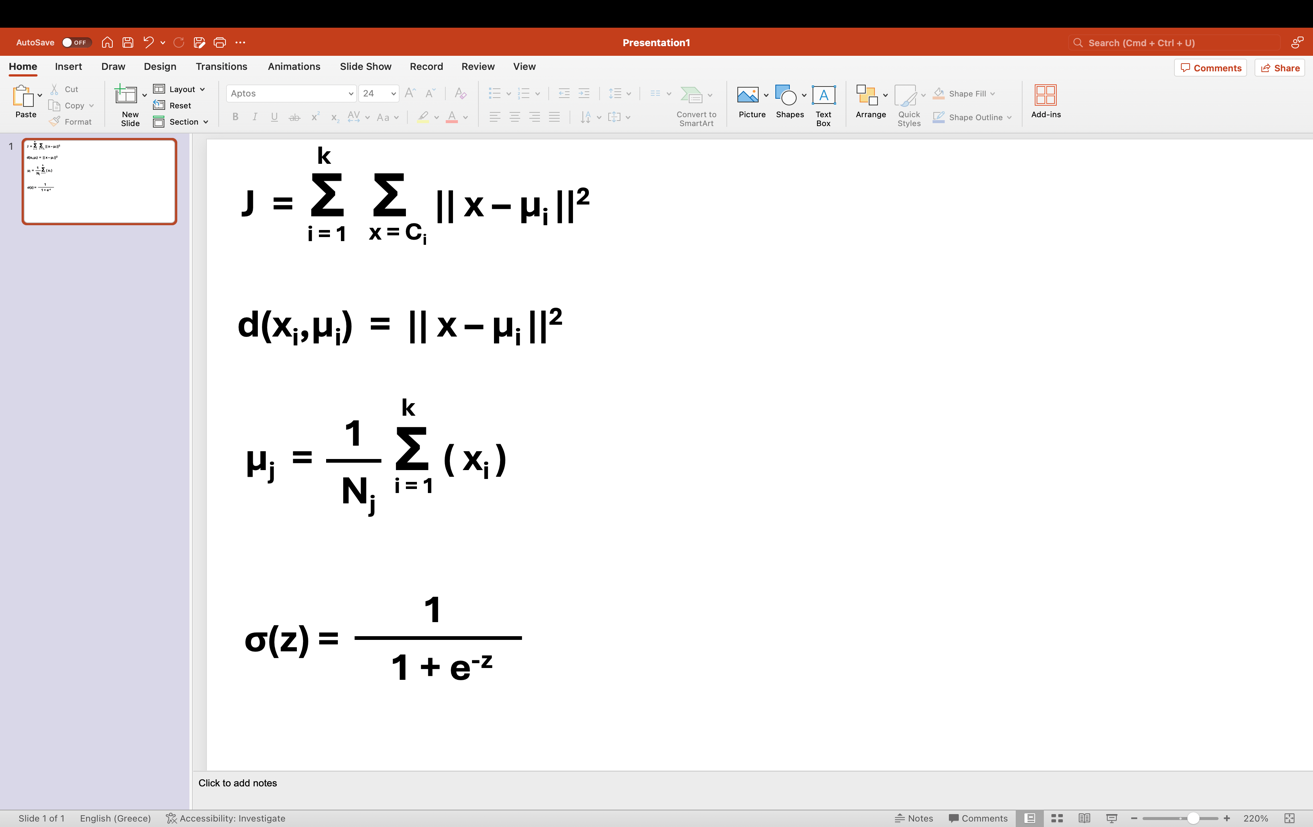
Also, consider that ***j = 1, 2, …, k***.



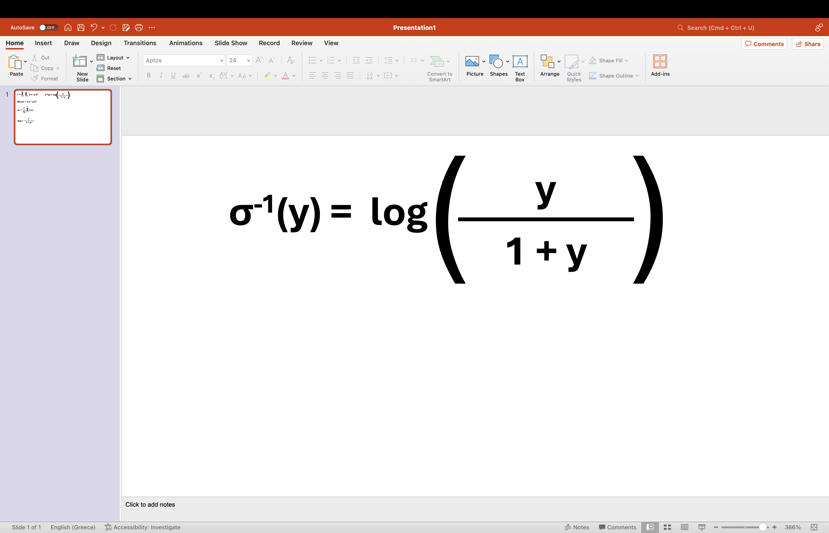
The mathematical expression for the cost function in k-means algorithm, where ***k*** represents the number of clusters & ***x*** is a data point. The clusters are expressed as ***Ci*** & their centroids as ***μi***.

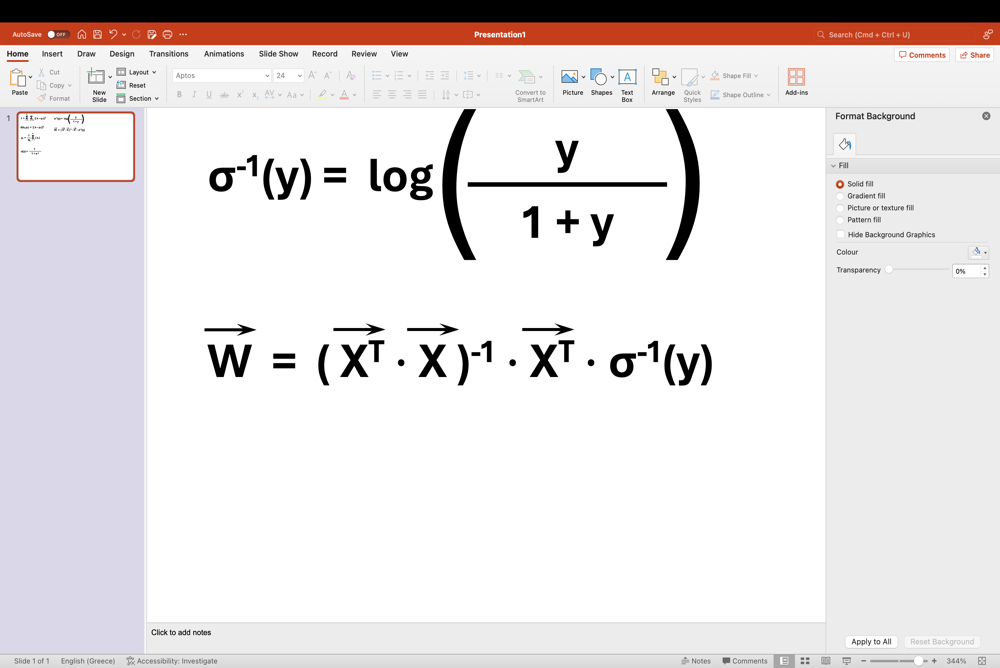
**TRAINING USING SIGMOID ACTIVATION**

The first layer of the model uses the Sigmoid Activation Function to train each neuron independently to the others. For that process, the Inverse Sigmoid is also used, as the model directly computes the weights using Least Squares Method.

 The sigmoid function, where ***z*** represents the sum of inputs ***X***, with weights ***W*** & bias ***b***.

The result can be any number from (0, 1), useful for applications in probability methods.

The inverse sigmoid, which turns a probability-type of value ***y***, which is the number from (0, 1), into a real number space, which can be infinity & that process is useful for linear methods.

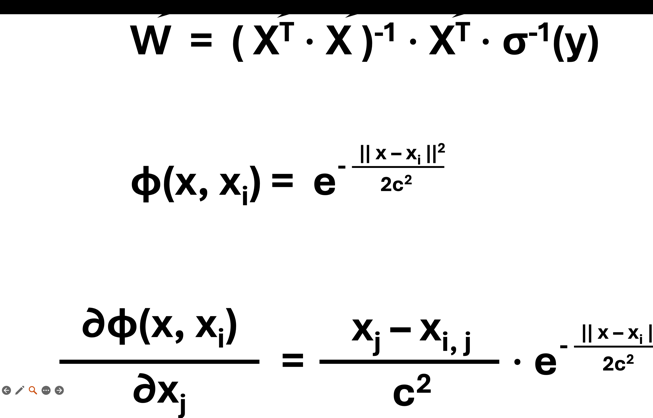


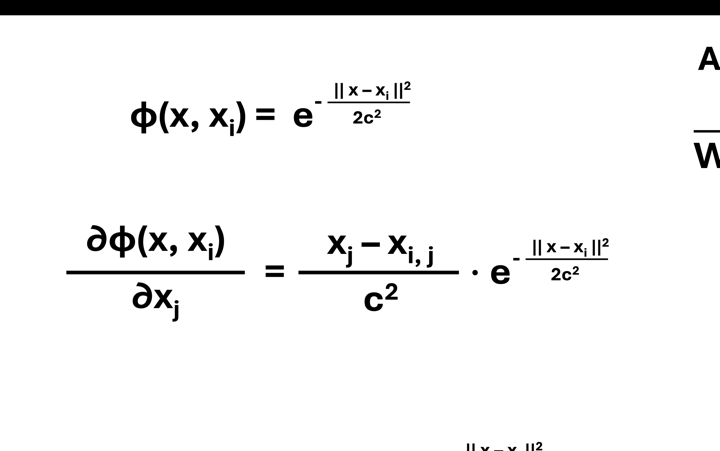
The Least Squares Method, to calculate the weights ***W***, directly from the inputs **X** & the computed **σ-1(y)**.

If any of the final outputs have infinite values or values that are not in the domain of the real numbers, they are just replaced in the vector ***W*** with zeros, for stability reasons.

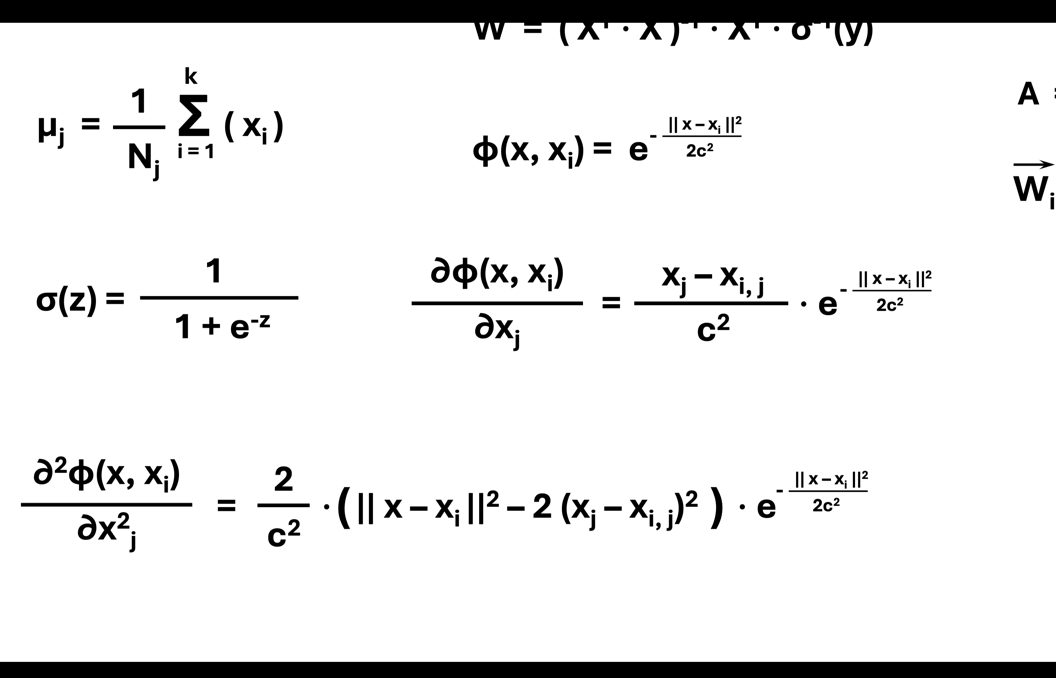
**TRAINING USING RBF WITH PARTIAL DIFFERENTIAL EQUATION CONSTRAINTS**

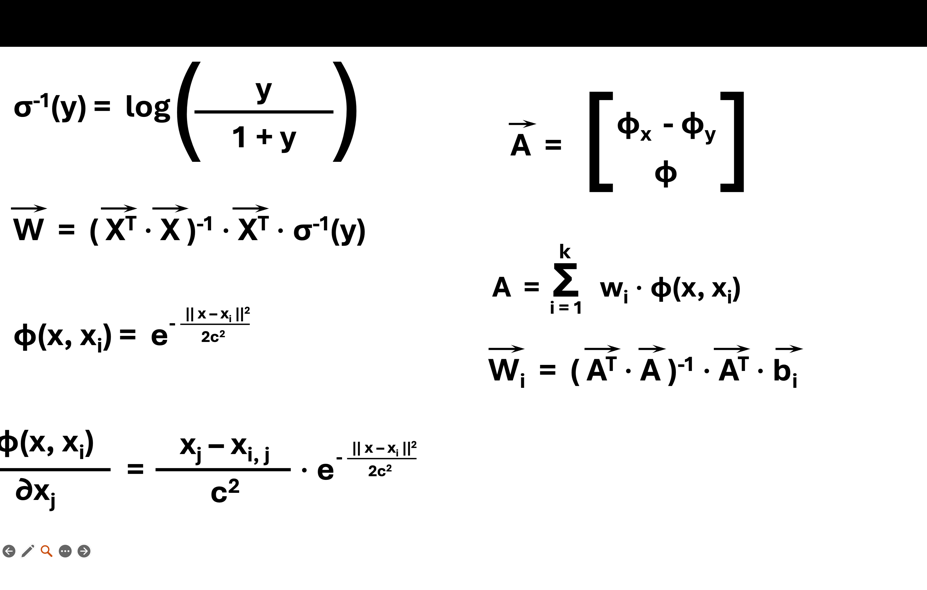
Additionally, the model uses the Radial Basis Function (RBF), which functions under PDE-based constraints. For that process, the algorithm firstly includes the calculation of the RBF-kernel & up to its second derivative. After that, the model can now construct the vector ***A*** for the RBF-kernel functions, using the calculated results from the derivatives, to finally conclude to the output weights, using the Least Squares Method.

The model computes the RBF-kernel & its derivatives, where ***φ*** represents the kernel, ***x*** refers to the data point that is inputted & ***xi*** refers to the current center / neuron, for ***i = 1, 2, …, k***, and width control parameter ***c***.

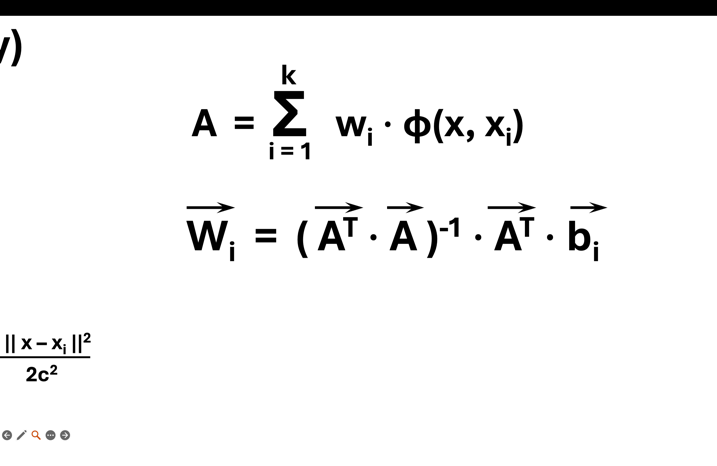


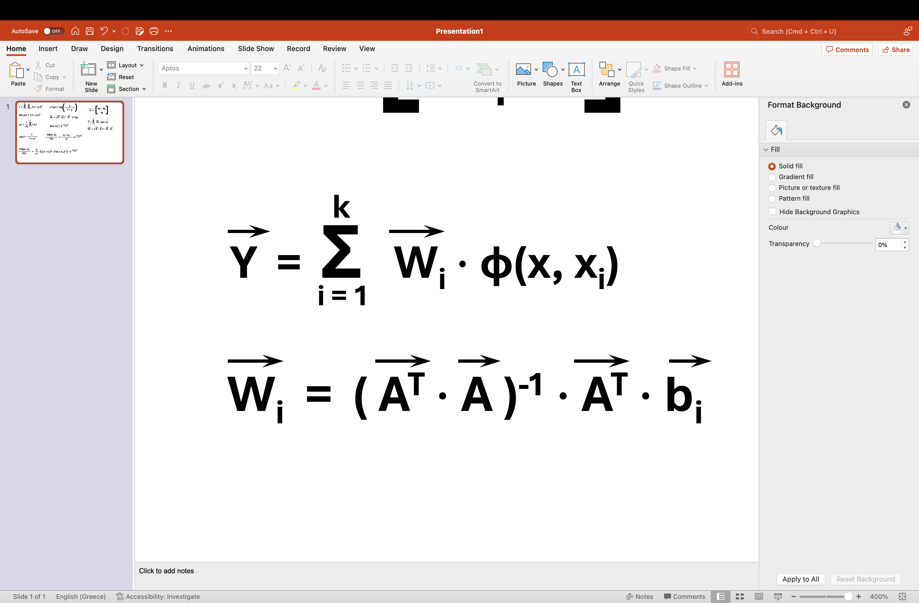
For the computation of the derivatives, we take into consideration all the elements of ***x*** & ***xi***.

We do those operations, with respect to both x & y, to find the gradients ***φx*** & ***φy***.



Then, the model uses the gradients & the kernel function, to conclude to the final vector for the kernels.

For every current point / neuron, a separate weights vector is computed this time.



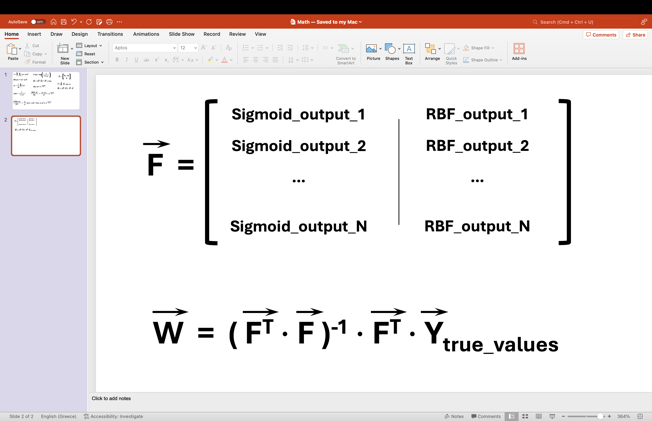
The final output for the RBF layer is computed as a result of the linear combination of the input data, applied to the kernel function.

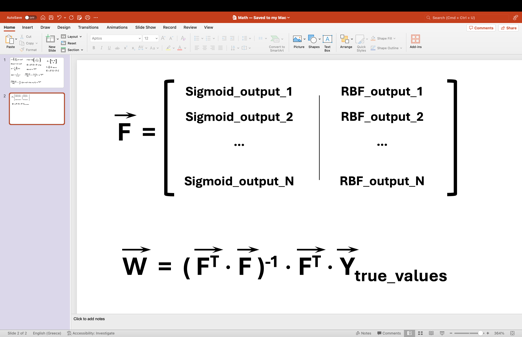
**APPLYING K-FOLD CROSS – VALIDATION**

Finally, the training process applies K-Fold Cross-Validation, in both Sigmoid & RBF networks, to ensure that the model would not become excessively dependent on any of the subsets of data used.

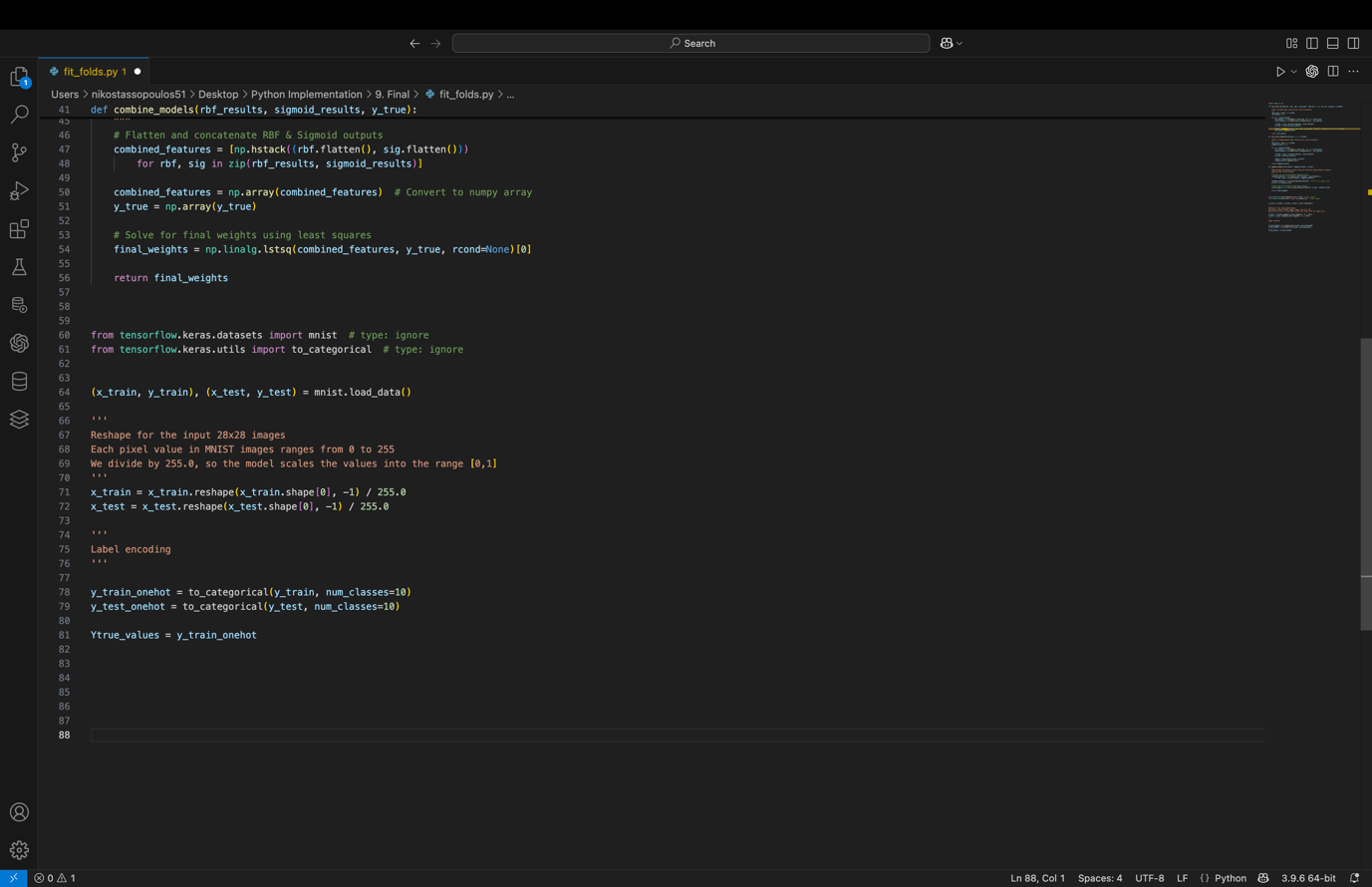
The main idea is that the dataset is spitted into k folds, ***D = D1, D2, …, Dk*** and each one of the folds contains ***N / k*** data points, where ***N*** is the total number of data points. The model is trained for each fold separately on a data subset, then another data subset is applied to it for validation.

The functions call the RBF & Sigmoid trainings accordingly, storing their results, for combination in a separate matrix, where each column represents the output elements extracted.



****

Where the vector ***Ytrue\_values*** contains the true values of the data points that have been imported from the dataset.

The true values from MNIST, can be imported as explained here:

**THE PREDICTION FUNCTIONS**

In this one, the model uses both Sigmoid & RBF predictions, using them for combination, based on Universal Approximation Theorem. Those predictions, which are in forms of vectors, are combined into a single vector, that includes all of the elements. Using that vector, final predictions are computed, using the prediction elements & the weights.

