**GSNR Prediction using Artificial Neural Networks (ANNs)**

# **Introduction**

## **Overview of the Problem**

Predicting General Signal-to-Noise Ratio (GSNR) is critical for optimizing the performance of communication systems. Accurate GSNR predictions can enhance signal quality, reduce errors, and improve overall system efficiency. This project aims to leverage artificial neural networks (ANNs) techniques to develop a robust model that can predict GSNR values based on various input features.

## **Description of the Dataset**

The dataset used in this project, European Topology 6 Paths, contains multiple features that influence the GSNR for 76 channels. It includes multiple features such as power, NLI (non-linear interference), ASE (amplified spontaneous emission), total distance, span and frequency. Each row in the dataset corresponds to a unique observation, providing a comprehensive set of data points for training and evaluating predictive models. The goal is to use this dataset to train various machine learning models and identify the one that delivers the most accurate GSNR predictions.

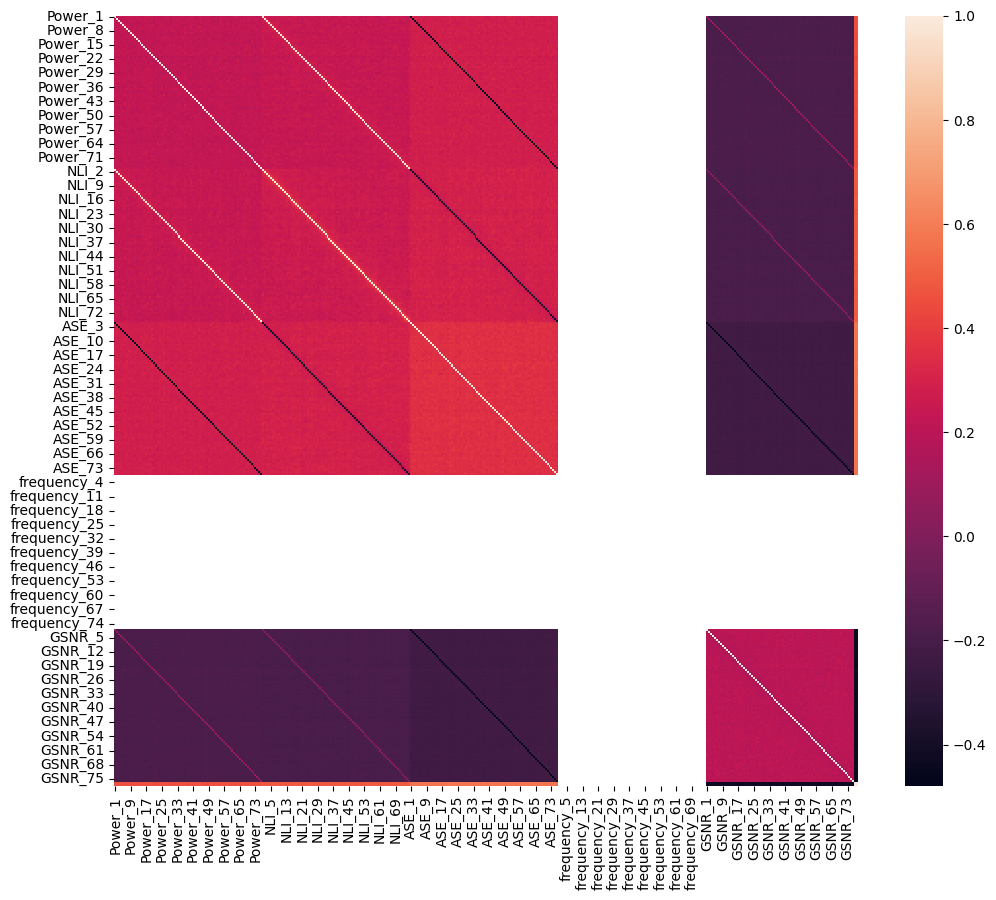
# **Understanding the Data**

We used correlation matrix heatmaps to explore the relationships between different features of the dataset. The provided heatmap of the entire dataset visually represents the correlations between all the features, which can help identify which features are strongly correlated with each other.

## **Correlation Matrix Heatmap**

The correlation matrix heatmap is a useful tool for visualizing the strength and direction of linear relationships between features. In the heatmap:

* **Color Intensity**: Indicates the strength of the correlation, with darker colors representing stronger correlations.
* **Positive Correlations**: Features that increase together are shown in shades of red.
* **Negative Correlations**: Features where one decreases as the other increases are shown in shades of blue or purple.

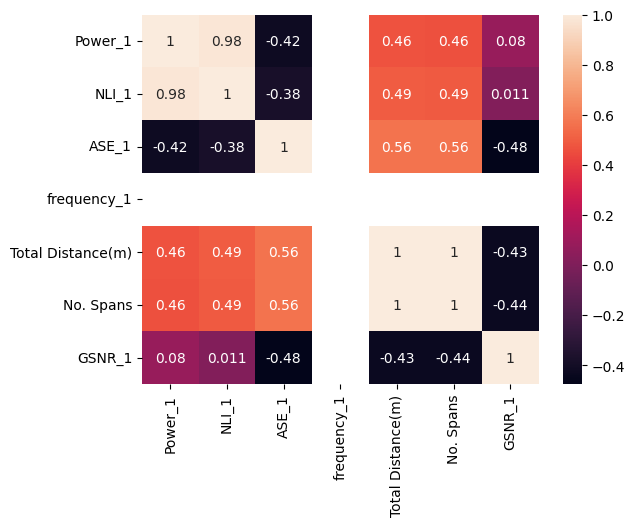


*Figure 1. Correlation Matrix Heatmap for Complete Dataset*

From the above plot we can see that frequency features have almost zero relation with other features. But still we cannot clearly see what is happening here. So for a better understanding, we will be looking at some different channels to get a better view.

## **Channel 1**

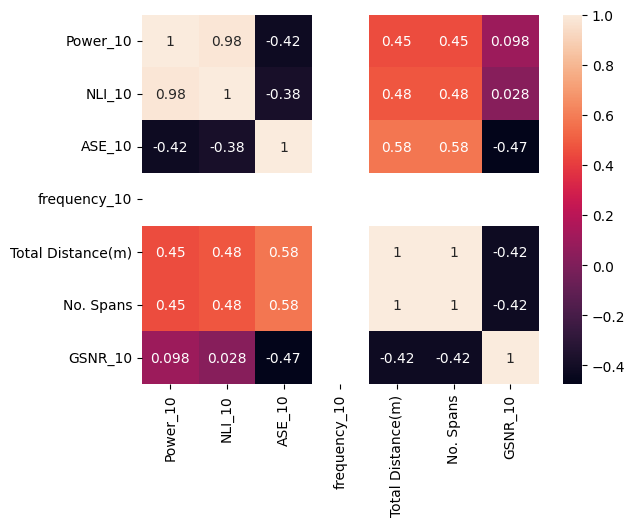
The following heatmap visualizes the correlation matrix for the dataset containing the columns 'Power\_1', 'NLI\_1', 'ASE\_1', 'frequency\_1', 'Total Distance(m)', 'No. Spans', and 'GSNR\_1'. Each cell in the heatmap represents the correlation coefficient between two variables, with values ranging from -1 to 1. Positive values indicate a direct relationship, while negative values indicate an inverse relationship. For example, 'Power\_1' and 'NLI\_1' have a high positive correlation (0.98), indicating that they vary together. On the other hand, 'ASE\_1' and 'GSNR\_1' have a negative correlation (-0.48), suggesting that as one increases, the other decreases. The heatmap uses color coding to visually represent the strength and direction of these correlations, making it easier to identify strong relationships between variables.



*Figure 2. Correlation Matrix Heatmap for Channel 1*

## **Channel 10**

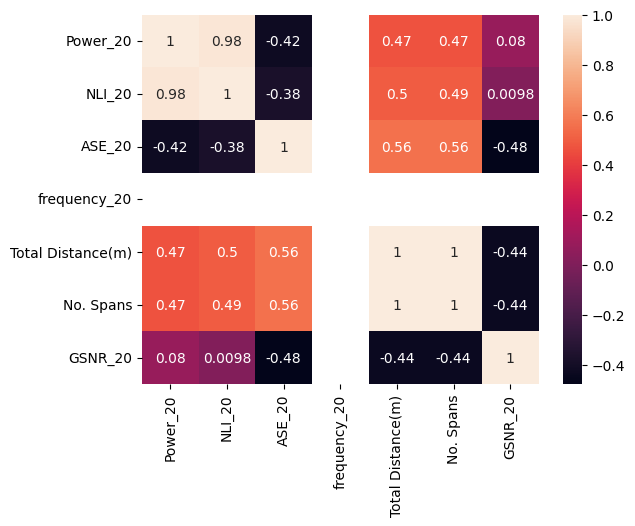
The following heatmap visualizes the correlation matrix for the dataset containing the columns 'Power\_10', 'NLI\_10', 'ASE\_10', 'frequency\_10', 'Total Distance(m)', 'No. Spans', and 'GSNR\_10'. Each cell in the heatmap represents the correlation coefficient between pairs of variables, with the values ranging from -1 to 1. Positive correlations indicate that as one variable increases, the other tends to increase, while negative correlations indicate an inverse relationship. For instance, 'Power\_10' and 'NLI\_10' have a high positive correlation of 0.98, suggesting a strong direct relationship. Conversely, 'ASE\_10' and 'GSNR\_10' have a negative correlation of -0.47, indicating an inverse relationship. The color intensity in the heatmap provides a visual cue for the strength and direction of these correlations, making it easier to quickly identify strong positive or negative relationships between variables.



*Figure 3. Correlation Matrix Heatmap for Channel 10*

## **Channel 20**

The heatmap visualizes the correlation matrix for the dataset containing the columns 'Power\_20', 'NLI\_20', 'ASE\_20', 'frequency\_20', 'Total Distance(m)', 'No. Spans', and 'GSNR\_20'. Each cell in the heatmap represents the correlation coefficient between two variables, with values ranging from -1 to 1. Positive values indicate a direct relationship, while negative values indicate an inverse relationship. For example, 'Power\_20' and 'NLI\_20' have a high positive correlation (0.98), indicating that they vary together. On the other hand, 'ASE\_20' and 'GSNR\_20' have a negative correlation (-0.48), suggesting that as one increases, the other decreases. The heatmap uses color coding to visually represent the strength and direction of these correlations, making it easier to identify strong relationships between variables.



*Figure 4. Correlation Matrix Heatmap for Channel 20*

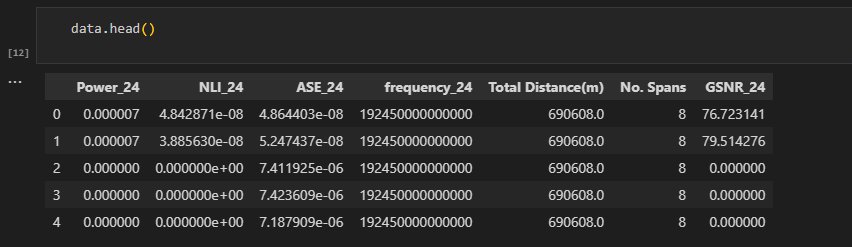
From the above different channel correlation plots, we can conclude that the frequency feature has almost zero significance which means we can discard this feature. But, before we discard any feature, we would like to apply different models and then we will decide whether feature selection is applicable or not.

# **Data Preprocessing**

For our prediction task, we used channel 24 (gsnr\_24) as the target variable.

## **Examine first few rows and column types**

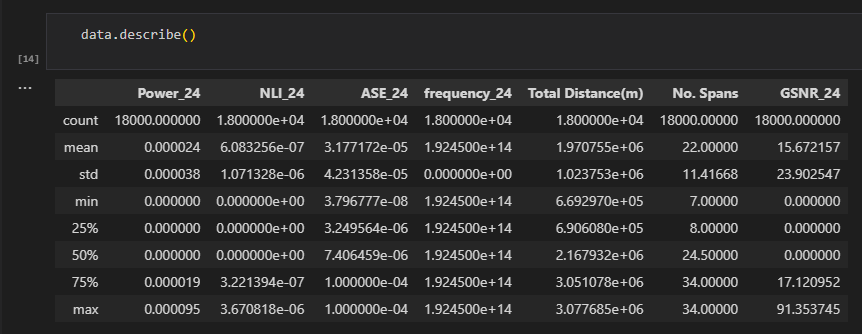
We started by examining the first few rows of the dataset using the data.head() method to get an initial overview of the data. All the columns were float64 except frequency and number of spans that were int.



*Figure 5. First few rows of data*

## **Summary Statistics and look for null values**

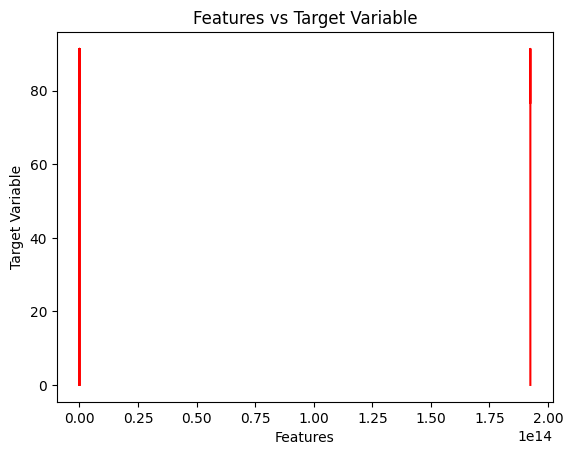
Summary statistics were generated using data.describe(), which provided insights into the central tendency, dispersion, and overall distribution of the data. We also check for null values but there were no null values at all.



*Figure 6. Summary statistics of data*

## **Visualizing gsnr\_24**

We visualize the data using pyplot module from the matplotlib library. The following figure shows that the data is non-linear which means we will be applying only non-linear supervised algorithms for making predictions.

****

*Figure 7. Visual representation of data*

# **Data Splitting**

To evaluate the performance of our predictive models effectively, we split the dataset into training and test sets. This approach helps us to understand how well the models generalize to unseen data.

We used the train\_test\_split function from the scikit-learn library to split the data, with 80% of the data allocated to the training set and 10% each to the validation set and test set. The random state was set to 42 to ensure reproducibility of the results.

Here are the details of the split:

* **X\_train size**: (14400, 6) – This subset contains 14,400 samples with 6 features each, used to train the models.
* **y\_train size**: (14400,) – This subset contains 14,400 target values corresponding to the training samples.
* **X\_val size**: (1800, 6) – This subset contains 1800 samples with 6 features each, used to validate the models.
* **y\_val size**: (1800,) – This subset contains 1800 target values corresponding to the validation samples.
* **X\_test size**: (3600, 6) – This subset contains 3,600 samples with 6 features each, used to test the models.
* **y\_test size**: (3600,) – This subset contains 3,600 target values corresponding to the test samples.

# **Feature Scaling**

Feature scaling is a crucial preprocessing step in many machine learning workflows. It ensures that all features contribute equally to the model's performance by putting them on a similar scale. This is particularly important for algorithms sensitive to the magnitude of feature values, such as Support Vector Machines, k-Nearest Neighbors, and neural networks.

In this task, we used the StandardScaler from the scikit-learn library to perform feature scaling. The StandardScaler standardizes features by removing the mean and scaling to unit variance, which transforms the data to have a mean of 0 and a standard deviation of 1.

By scaling the features, we enhance the performance of our machine learning models, ensuring they converge faster and perform better by treating all features equally, regardless of their original scale.

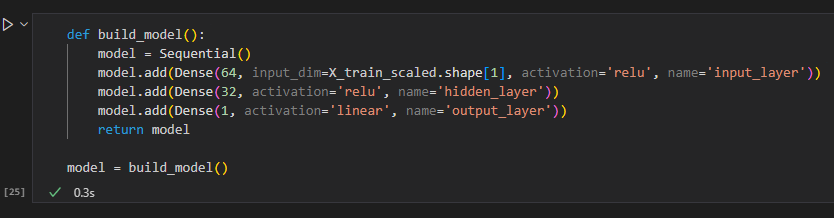
# **Implementation of Artificial Neural Networks (ANNs)**

In machine learning, a neural network (also artificial neural network or neural net, abbreviated ANN or NN) is a model inspired by the structure and function of biological neural networks in animal brains. Here, we will build different ANN models and will try to find the best suited model for the given regression task.

## **Simple Model**

The network architecture includes:

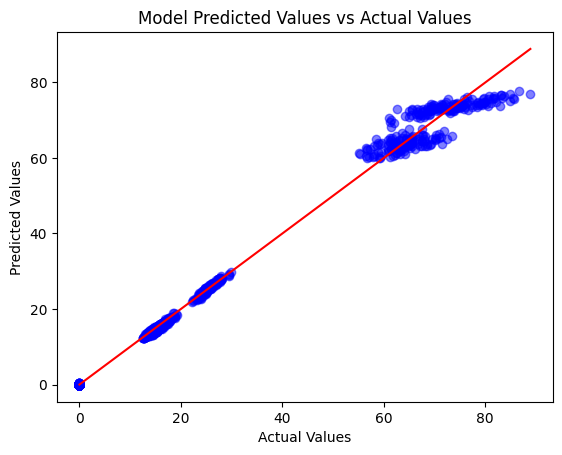
* An input layer with 64 neurons and ReLU activation.
* A hidden layer with 32 neurons and ReLU activation.
* An output layer with a single neuron and linear activation.



The model is compiled with the Adam optimizer and mean squared error (MSE) loss function. Training is performed for 100 epochs with a batch size of 32, using validation data to monitor performance.

After training, the model is evaluated on a test set, achieving a **test loss of 2.451** and a **mean absolute error (MAE) of 0.677**.

The following plot visualizes the model's predictions against the actual values, showing a strong alignment along the line of perfect prediction (red line). The clustering of points around the line indicates good predictive performance.



# **Hyperparameter Tuning**

In machine learning, a hyperparameter is a parameter, such as the learning rate or choice of optimizer, which specifies details of the learning process, hence the name hyperparameter. This is in contrast to parameters which determine the model itself.

Here will be looking for finding the best model with optimal parameters to do the job perfectly.

## **Hyperparameter Tuning without Regularization and Early Stopping**

To find the best hyperparameters for a neural network model. Below is a breakdown of the steps and the results:

### **Imports and Setup**

The necessary libraries, including Keras Tuner, are imported to facilitate the hyperparameter tuning process.

!pip install keras-tuner

import keras\_tuner as kt

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

import tensorflow as tf

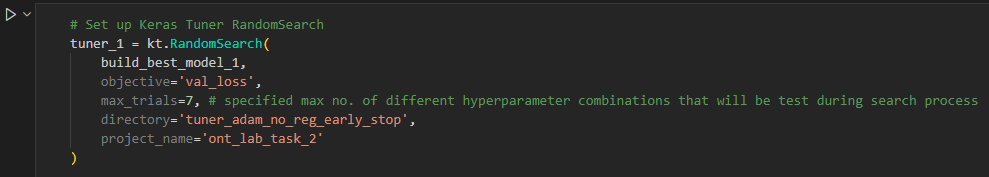
### **Model Building Function**

This function defines a neural network with:

* One input layer.
* Three hidden layers, where the number of units and dropout rates are hyperparameters to be tuned.
* One output layer.
* The optimizer's learning rate is also a hyperparameter to be tuned.

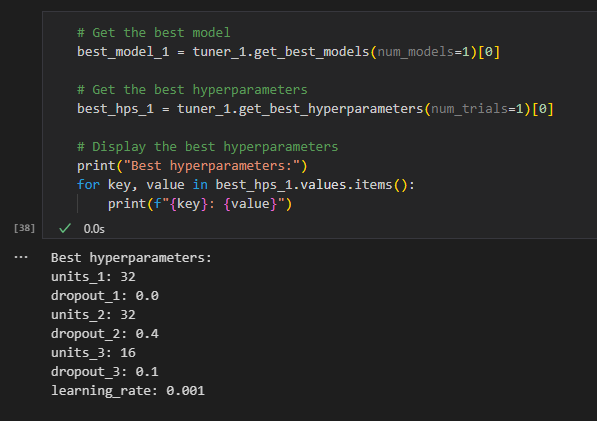
### **Setting Up Keras Tuner**

Keras Tuner's RandomSearch is configured to optimize the model based on validation loss, testing up to 7 different combinations of hyperparameters.



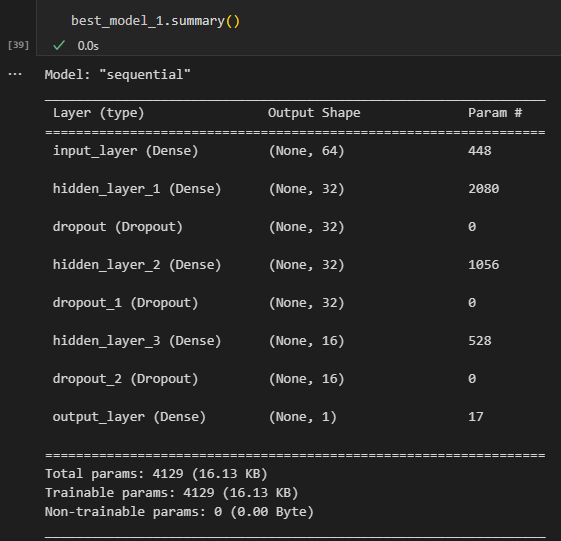
### **Hyperparameter Search, Retrieve Best Model and Hyperparameters**

The tuner searches for the best hyperparameters using the training and validation data. The best model and its hyperparameters are retrieved and printed. We can see in the following screenshot the optimal parameters found during the tuner search.



### **Model Summary**

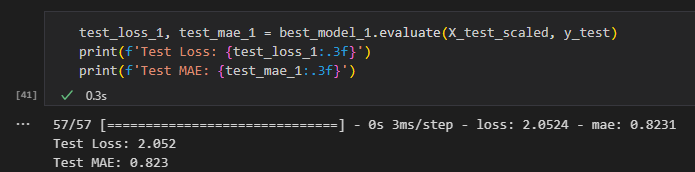
Following is the summary of the model found during the tuner search while finding the optimal parameters:



The model, named "sequential," comprises several layers: an input layer, three hidden layers with dropout regularization, and an output layer. Specifically, the input layer has 64 units, followed by three hidden layers with 32, 32, and 16 units respectively. Each hidden layer is accompanied by a dropout layer with dropout rates of 0.0, 0.4, and 0.1. The output layer consists of a single unit. The model has a total of 4129 trainable parameters. This architecture aims to balance model complexity and regularization to achieve optimal performance on the given task.

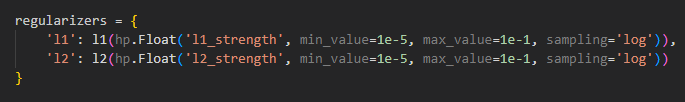
### **Model Evaluation**

After training, the model is evaluated on a test set, achieving a **test loss of 2.052** and a **mean absolute error (MAE) of 0.823**.



## **Hyperparameter Tuning with Regularization and Early Stopping**

I applied hyperparameter tuning with L1 and L2 regularization using various values and incorporated early stopping to prevent overfitting.



However, the results obtained with these additional techniques seem weaker compared to the previous model found without regularization and early stopping. Specifically, the initial hyperparameter tuning without regularization and early stopping yielded a test loss of 2.052 and a test MAE of 0.823. In contrast, the model tuned with regularization and early stopping resulted in a significantly higher test loss of 6.294 and a test MAE of 1.293. This indicates that the regularization techniques and early stopping did not improve the model performance in this case and might have led to underfitting.

|  |  |  |
| --- | --- | --- |
|  | Before | After |
| Test Loss | **2.052** | **6.294** |
| Test MAE | **0.823** | **1.293** |

# **Conclusion**

In our experimentation with Artificial Neural Networks (ANNs), we followed a systematic approach to optimize model performance. Initially, we implemented a simple ANN model with three layers consisting of 64, 32, and 1 units respectively. The activation functions used were ReLU for the first two layers and linear for the output layer. This model yielded a test loss of 2.451 and a test MAE of 0.677.

Next, we performed hyperparameter tuning without any regularization or early stopping, which led us to a more complex model. This optimized model included an input layer with 64 units, three hidden layers with 32, 32, and 16 units respectively, and dropout layers for regularization. The architecture was as follows:

* **Input Layer**: 64 units
* **Hidden Layer 1**: 32 units, Dropout (0.0)
* **Hidden Layer 2**: 32 units, Dropout (0.4)
* **Hidden Layer 3**: 16 units, Dropout (0.1)
* **Output Layer**: 1 unit

This model achieved improved performance with a test loss of 2.052 and a test MAE of 0.823.

Finally, we applied hyperparameter tuning with L1 and L2 regularization and early stopping to further enhance the model. Contrary to expectations, the resulting model's performance deteriorated, with a test loss of 6.294 and a test MAE of 1.293.

These results suggest that the **best model for our task** was achieved through **hyperparameter tuning without regularization and early stopping**. The added complexity of regularization and early stopping did not improve the model's performance and instead led to underfitting. Therefore, the optimal model for our data and task is the one obtained from hyperparameter tuning without regularization and early stopping.

**THE END**