

Fast Machine Learning Surrogate Model for Complex Physical Systems

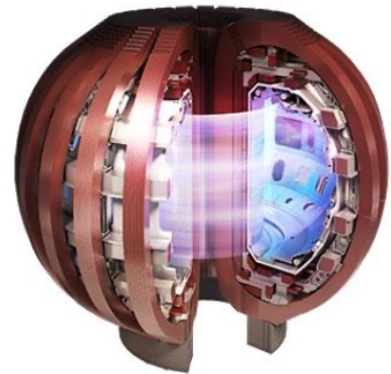
Project overview

Many complex physical and engineering systems are characterized by **high-dimensional parameter spaces** and **computationally expensive simulations**, making exhaustive exploration infeasible. In such settings, **surrogate models** provide a practical alternative by approximating costly simulations with fast, data-driven predictors, enabling rapid evaluation and decision-making under tight computational constraints.

Fusion plasma modeling presents a representative example of this problem: system behavior depends on nonlinear interactions among multiple parameters, while **high-fidelity simulations can take hours to days per run**. This severely limits iteration speed and parameter-space exploration.

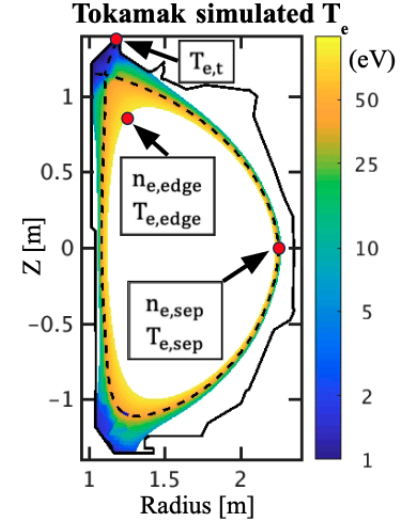
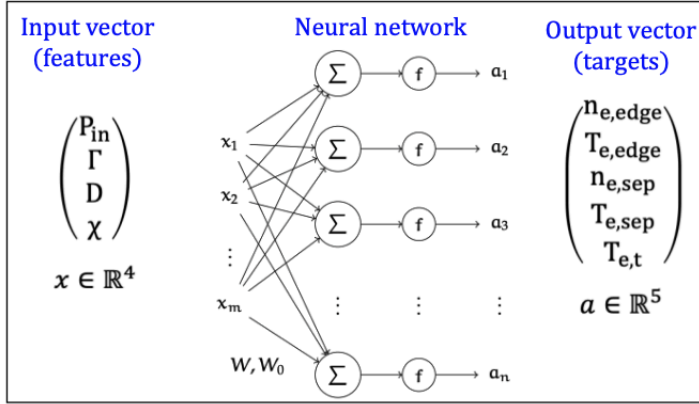
To address this, I trained a **feedforward neural network** on simulation data from a tokamak plasma system, to predict key system responses under previously unseen conditions. The resulting surrogate model achieves **high out-of-sample accuracy** while **reducing evaluation time from hours to milliseconds**. I benchmarked its performance against simpler regression models to quantify tradeoffs between predictive power, interpretability, and computational cost.

Example of a Tokamak



Data Generation and Preprocessing

This project adopts a **supervised learning framework**, in which the model learns to map input features to output targets using historical data. The inputs correspond to controllable system parameters, while the outputs represent key system response metrics. **Input features** include injected power (P_{in}), gas flux (Γ), particle diffusivity (D) and heat diffusivity (χ). **Target variables** consist of plasma density (n_e) and temperature (T_e) at two spatial locations (edge and separatrix), along with the peak temperature at the wall target plate ($T_{e,t}$).



The dataset consists of **1,024 simulated plasma states**, generated using a high-fidelity numerical code for modeling plasma behavior near the tokamak boundary. Input parameters were sampled over the predefined ranges (injected power P_{in} from 2 to 20 MW, gas flux Γ from 10^{20} to 10^{22} atoms/s, particle diffusivity D from 0.1 to 2 m^2/s , and heat diffusivity χ from 0.3 to 6 m^2/s) using a **Sobol low-discrepancy sequence** to ensure uniform coverage of the four dimensional parameter space.

All data processing and model training were performed in **Python 3.10**, using standard scientific and machine-learning libraries (*NumPy*, *SciPy*, *pandas*, *matplotlib*, *TensorFlow*). The full dataset was stored in a **pandas DataFrame**, enabling efficient filtering, validation, and downstream analysis.

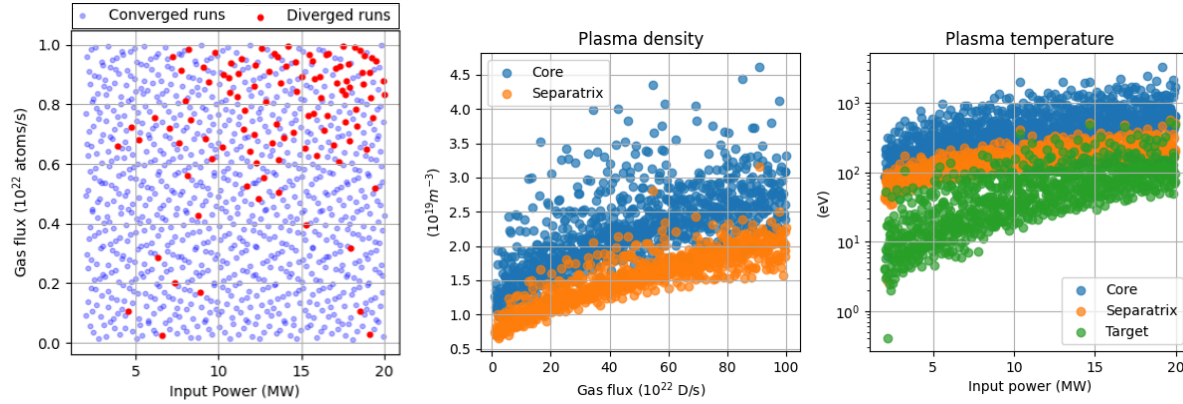
	power	flux	D	Chi	convMetric	runTime	necore	tecure	neompsep	teompsep	tetpeak
1.0	2.03	76.74	0.74	3.57	0.013357	0.99990	2.421595	87.753569	1.668311	44.285957	4.071856
2.0	18.79	9.48	1.45	1.51	0.028593	0.99990	1.617890	1295.978818	1.051478	353.558060	294.988417
3.0	13.21	66.93	0.34	4.93	0.452556	0.10295	3.503367	353.001058	2.310341	314.932087	4263.070839
4.0	10.00	49.08	1.55	2.69	0.013265	0.99990	2.115895	415.895896	1.521478	146.283063	36.095903
5.0	6.54	54.15	1.26	2.27	0.014374	0.99990	2.206144	319.063417	1.583377	117.411926	19.711827

Simulation outputs were screened for **numerical convergence** and **physical plausibility** prior to model training. A simulation was retained only if it satisfied the following criteria:

- The separatrix temperature variation in the final 0.1 s (“convMetric”) was < 5%;
- The total simulation time (“runTime”) exceeded 0.95 s;
- The peak wall target temperature did not exceed the core temperature, which would be physically impossible.

After applying these filters, **920 simulations ($\approx 90\%$)** were retained for model development.

Before training, I conducted **sanity checks** to confirm that the data obeyed expected physical relationships, for example, plasma density increasing with gas flux, temperature increasing with injected power, and consistent co-variation of temperature and density across spatial regions. These checks confirmed that the dataset was coherent and suitable for supervised learning.



Finally, the dataset was split into disjoint **training, validation, and test sets (70% / 15% / 15%)**. All input features were **standardized** (zero mean, unit variance) to improve numerical conditioning and stabilize gradient-based optimization. Scaling parameters were computed exclusively from the training set and applied consistently to validation and test data.

The output variables exhibited significant scale separation: temperature targets span several orders of magnitude ($\approx 10^2$ – 10^3) relative to densities ($\approx 10^0$). To mitigate variance instability and prevent scale dominance in the loss function, **temperature outputs were log-transformed**, while densities were kept in linear scale. All targets were then **standardized** (zero mean, unit variance), again using training-set statistics only, ensuring balanced contributions across outputs during training.

Model Architecture and Training Framework

A configurable **feedforward neural network** was implemented in Python using **TensorFlow/Keras**. I developed a modular training function that builds, compiles, and fits the model using a dictionary of hyperparameters, including the number of layers (num_layers), neurons per layer (units), activation function (activation), learning rate (learning_rate), batch size (batch_size), and an option for batch normalization (use_batch_norm). This setup enables systematic experimentation and fair comparison across model configurations.

The first part of the code builds the network architecture: the chosen activation function is applied to all hidden layers, while the **output layer uses a linear activation**, as the target variables are already normalized. The model is trained using the **Adam optimizer**, which adaptively adjusts the learning rate for each parameter, leading to faster and more stable convergence in high-dimensional problems.

To control overfitting, **early stopping** is employed based on validation loss, terminating training if no improvement is observed for **30 consecutive epochs**. The maximum number of training epochs is capped at 300.

```
# Set random seed for reproducibility
os.environ['PYTHONHASHSEED']=str(seed_choice)
random.seed(seed_choice)
np.random.seed(seed_choice) # Numpy
tf.random.set_seed(seed_choice) # TensorFlow

# Define the model with hyperparameters
model = keras.Sequential()

# Input layer
model.add(keras.layers.Input(shape=(x_train.shape[1],)))

# Hidden layers
for _ in range(hp['num_layers']):
    model.add(keras.layers.Dense(units=hp['units'], activation=hp['activation']))
    if hp.get('use_batch_norm', False):
        model.add(keras.layers.BatchNormalization())

# Output layer with linear activation (since targets are scaled)
model.add(keras.layers.Dense(units=y_train.shape[1], activation='linear'))

# Choose optimizer, then compile module
optimizer = tf.keras.optimizers.Adam(learning_rate=hp['learning_rate'])
model.compile(optimizer=optimizer, loss='mean_squared_error')

# Define Early Stopping callback (monitor validation loss)
early_stopping = EarlyStopping(monitor='val_loss', patience=max_patience, restore_best_weights=True)

# Train the model with hyperparameters
history = model.fit(x_train, y_train,
                    epochs=max_epochs, # Fixed number of epochs
                    batch_size=hp['batch_size'],
                    validation_data=(x_val, y_val),
                    callbacks=[early_stopping],
                    shuffle=False,
                    verbose=verbose)
```

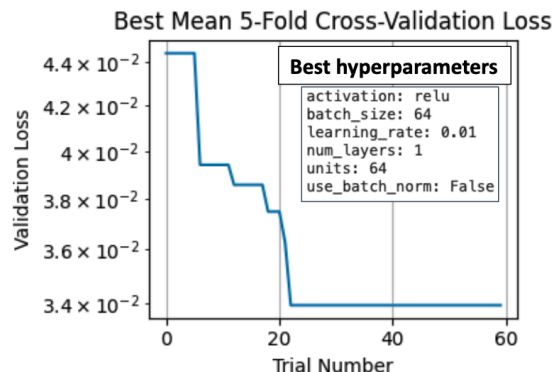
Hyperparameter Optimization Procedure

Model performance is sensitive to the choice of **hyperparameters** (including network depth, width, learning rate, and activation functions) making systematic tuning essential. To efficiently explore the resulting high-dimensional search space, I employed Bayesian optimization using the **Tree-Structured Parzen Estimator (TPE)**, combined with **5-fold cross-validation** to obtain robust out-of-sample estimates.

```
# Define the original choices of hyperparameter values
num_layers_choices = [1, 2]
units_choices = [16, 32, 64]
activation_choices = ['relu', 'tanh']
learning_rate_choices = [0.0001, 0.001, 0.01]
batch_size_choices = [16, 32, 64]
use_batch_norm_choices = [True, False]
```

A **constrained hyperparameter space** was defined based on standard practice in applied machine learning and the limited dataset size, in order to avoid excessive model capacity. For each TPE trial, a candidate hyperparameter configuration was evaluated via 5-fold cross-validation on the **782 non-test samples**: the model was trained on four folds and validated on the remaining fold, rotating over all folds. The **mean validation loss** across folds was used as the optimization objective.

The procedure was run for **60 TPE trials**. The optimization trace shows that the objective stabilizes after approximately **25 trials**, indicating effective convergence of the search. The optimal configuration corresponds to a network with **one hidden layer of 64 units**, totaling **645 trainable parameters**. In each cross-validation fold, the model is trained on **626 samples**, resulting in a parameter-to-sample ratio of approximately **1.03**. While neural networks do not follow a strict parameter-to-sample rule, this ratio is generally considered conservative when combined with cross-validation and early stopping.

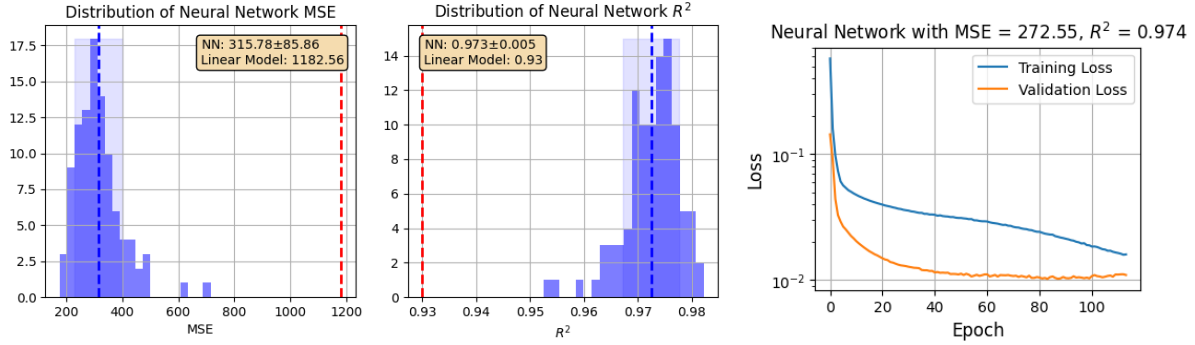


ML Model Evaluation

Neural network training is inherently **stochastic** due to factors such as random weight initialization, mini-batch ordering, early stopping sensitivity to validation-loss noise, and minor GPU numerical nondeterminism. As a result, final model weights and performance metrics exhibit non-negligible variability. To quantify this effect, I fixed the hyperparameters selected via cross-validation and **retrained the network 100 times using different random seeds**, which control initialization and data shuffling. Each trained model was evaluated on a held-out **test set of 138 samples**, yielding an ensemble of independently trained networks and allowing estimation of performance variation.

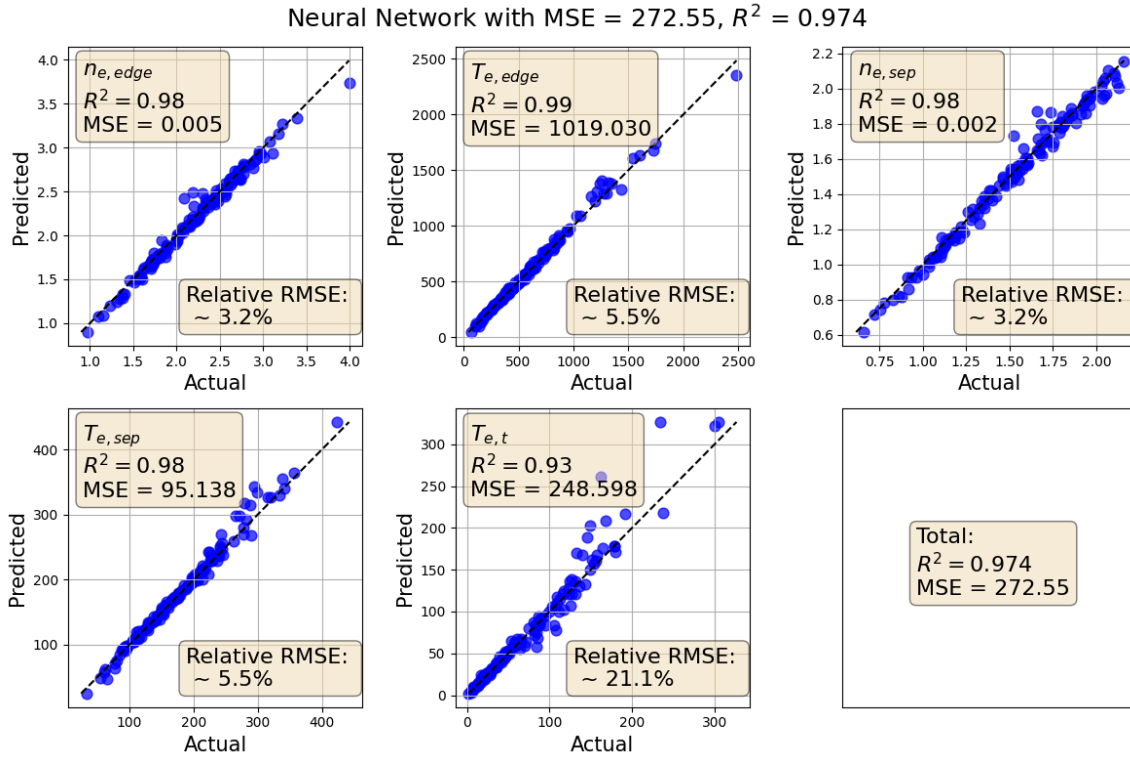
Model performance was assessed using two standard regression metrics: **Mean Squared Error (MSE)** and the **coefficient of determination (R^2)**. MSE measures the average squared prediction error and is sensitive to large deviations, while R^2 quantifies the fraction of variance in the target variables explained by the model. All metrics are computed **after inverse-transforming predictions back to physical units**. Because temperature targets span much larger numerical ranges than densities, raw MSE values are not directly comparable across outputs. To address this, an additional **relative RMSE** metric is reported, defined as RMSE normalized by the mean of the target variable. This provides a **scale-free measure of prediction error**, enabling meaningful comparison across targets.

The histograms below show the distribution of MSE and R^2 across the 100 training runs. Both metrics indicate a clear performance advantage over the linear regression baseline (discussed in the next section). Specifically, **the neural network achieves a substantially lower MSE (315.78 vs. 1182.56) and a higher R^2 (0.973 vs. 0.934) than the linear model**. The linear-model metrics lie several standard deviations away from the neural-network ensemble means, confirming that the observed performance gains are statistically robust.



The histograms summarize **aggregate performance across all targets**. To examine predictive behavior at the level of individual outputs, I selected a representative network from the ensemble with metrics close to the ensemble mean (MSE = 272.55, $R^2 = 0.974$). The corresponding training and validation loss curves show stable convergence and good generalization, with early stopping triggered by a lack of improvement in validation loss over the final 30 epochs.

Predicted versus true values are then compared for each of the five target variables. Both visual inspection and quantitative metrics indicate **high predictive accuracy for temperature and density outputs**, with the exception of the divertor target temperature $T_{e,t}$, which is the most challenging quantity to predict¹.



¹ This is physically expected: in tokamak plasmas, target-region temperatures are strongly influenced by localized, nonlinear edge and wall-interaction processes, making them significantly difficult to model.

Comparative Analysis: Neural Network vs. Linear Model

As a baseline, I trained a **linear regression model** using the same input features. Because plasma temperature and density are strictly non-negative, the regression was performed in **log-log space** to avoid unphysical negative predictions and to capture potential multiplicative (**power-law**) relationships between variables.

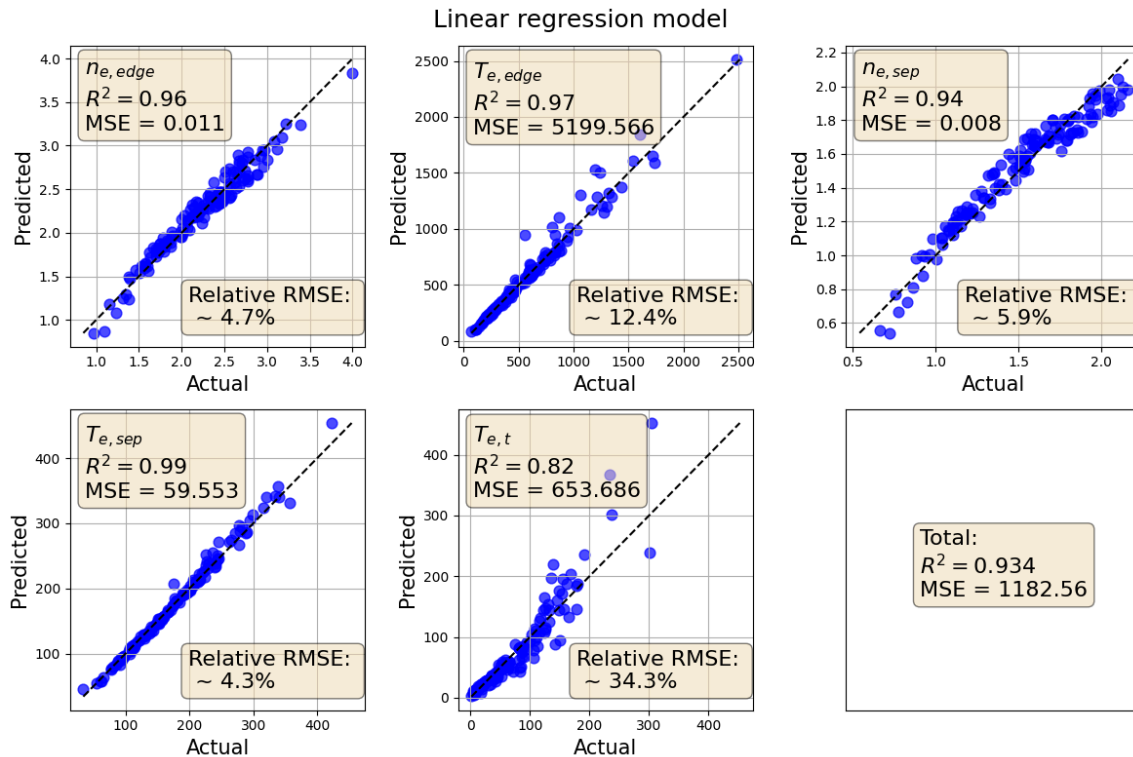
```
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import numpy as np

# Fit the linear regression model to the training data
linear_model = LinearRegression()
linear_model.fit(np.log(x_train), np.log(y_train))

# Make predictions on the test data
y_predicted_test_linear = np.exp(linear_model.predict(np.log(x_test)))

# Calculate performance metrics (MSE and R-squared)
mse_test_linear = mean_squared_error(y_test, y_predicted_test_linear)
r2_test_linear = r2_score(y_test, y_predicted_test_linear)
```

As previously anticipated, the **neural network outperforms the linear model**, achieving higher R^2 and lower MSE and relative RMSE. Performance for the separatrix temperature ($T_{e,sep}$) is broadly comparable between the two models, suggesting a weaker degree of nonlinearity for this quantity. As with the neural network, the linear model shows its weakest performance for the **divertor temperature** $T_{e,t}$, reflecting the intrinsic difficulty of modeling plasma behavior in the near-wall region.



Discussion and Conclusion

This project demonstrates that **a modestly sized neural network can act as an effective surrogate model for computationally expensive plasma simulations** when model capacity is controlled and performance is evaluated rigorously. By combining cross-validation, Bayesian hyperparameter optimization, and ensemble-based evaluation, the **neural network consistently outperforms a linear regression baseline**, achieving an approximately **70–75% reduction in mean squared error** and an increase in R^2 **from ~ 0.93 to ~ 0.97** on held-out test data.

The model's weakest performance is observed for the divertor temperature, a quantity known in plasma physics to be governed by highly localized and strongly nonlinear edge processes. This suggests that the remaining errors are driven primarily by **intrinsic physical complexity** rather than by overfitting or numerical instability.

Overall, this work illustrates how machine-learning surrogates can deliver **orders-of-magnitude speedups over full simulations of complex physical models** while preserving predictive accuracy for most quantities of interest, making them suitable for rapid parameter exploration and sensitivity analysis.

Based on this analysis, several directions could further improve performance:

- Expanding the simulation dataset to improve coverage of plasma regimes
- Enriching the feature set to capture additional physical dependencies
- Reparameterizing inputs and outputs using more physically meaningful quantities (e.g., power or particle fluxes)