Project Documentation

Power Plant Energy Output Prediction

Using Artificial Neural Networks (PyTorch)

ANN Regression | *Deep Learning* | *Combined Cycle Power Plant*

1. Introduction

1.1 Problem Statement

Predict the net hourly electrical energy output (PE) of a Combined Cycle Power Plant (CCPP) based on ambient environmental conditions. This is a regression task where the goal is to estimate a continuous target variable.

1.2 Objective

Build and train an Artificial Neural Network (ANN) using PyTorch to accurately predict the energy output given four input features: Temperature, Exhaust Vacuum, Ambient Pressure, and Relative Humidity.

1.3 Tools & Technologies

Tool	Purpose
Python 3.x	Programming language
PyTorch	Deep learning framework
Pandas	Data loading & manipulation
NumPy	Numerical operations
scikit-learn	Preprocessing & evaluation metrics
Matplotlib	Visualization
Jupyter Notebook	Development environment

2. Dataset Description

File: powerplant_data.csv

Records: 9,568 observations

Source: Combined Cycle Power Plant dataset

2.1 Feature Details

Column	Full Name	Unit	Role
AT	Ambient Temperature	°C	Input Feature
V	Exhaust Vacuum	cm Hg	Input Feature
AP	Ambient Pressure	millibar	Input Feature
RH	Relative Humidity	%	Input Feature

PE	Produced Energy (Net hourly electrical energy output)	MW	Target Variable

2.2 Data Quality

- Missing values: None (verified with df.isnull().sum())
- Data types: All columns are numeric (float64)

3. Data Preprocessing

3.1 Feature-Target Split

```
X = df.drop("PE", axis=1)   # Features: AT, V, AP, RH
y = df["PE"]                  # Target: PE
```

3.2 Train-Test Split

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

- Training set: 80% (~7,654 samples)
- Test set: 20% (~1,914 samples)
- Random state: 42 (for reproducibility)

3.3 Feature Scaling

Standard scaling (zero mean, unit variance) is applied using StandardScaler:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)   # fit on train only
X_test_scaled = scaler.transform(X_test)          # transform test using train stats
```

Why StandardScaler? Neural networks converge faster and perform better when input features are on a similar scale. StandardScaler normalizes each feature to have mean=0 and std=1.

3.4 Tensor Conversion & DataLoader

```
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.float32).view(-1, 1)

train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

- Data is converted from NumPy/Pandas to PyTorch tensors
- Target is reshaped to (n, 1) for compatibility with the model output
- DataLoader handles batching (batch size = 32) and shuffling

4. Model Architecture

4.1 Network Design

```
???????????????????????????????
?       Input Layer           ?
?       (4 neurons)           ?
?   AT, V, AP, RH             ?
???????????????????????????????
           ?
???????????????????????????????
?    Hidden Layer 1           ?
?    Linear(4, 6) + ReLU      ?
?    (6 neurons)              ?
???????????????????????????????
           ?
???????????????????????????????
?    Hidden Layer 2           ?
?    Linear(6, 6) + ReLU      ?
?    (6 neurons)              ?
???????????????????????????????
           ?
???????????????????????????????
?      Output Layer           ?
?      Linear(6, 1)           ?
?    (1 neuron ? PE)          ?
???????????????????????????????
```

4.2 Implementation

```
class ANN(nn.Module):
    def __init__(self):
        super(ANN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(4, 6),    # Hidden Layer 1
            nn.ReLU(),
            nn.Linear(6, 6),    # Hidden Layer 2
            nn.ReLU(),
            nn.Linear(6, 1),    # Output Layer
        )

    def forward(self, x):
        return self.model(x)
```

4.3 Design Choices

Choice	Value	Rationale
Hidden layers	2	Sufficient for learning non-linear patterns in tabular d
Neurons per hidden layer	6	Compact architecture to avoid overfitting on a small
Activation function	ReLU	Standard choice; avoids vanishing gradient problem
Output activation	None (linear)	Regression task requires unbounded continuous ou

4.4 Total Parameters

Layer	Parameters
Linear(4, 6)	4×6 + 6 = **30**
Linear(6, 6)	6×6 + 6 = **42**
Linear(6, 1)	6×1 + 1 = **7**
Total	**79**

5. Training

5.1 Configuration

Hyperparameter	Value

Loss Function	MSELoss (Mean Squared Error)
Optimizer	Adam (default lr=0.001)
Epochs	100
Batch Size	32

5.2 Training Loop

For each epoch:

- Training phase (model.train()):
- Iterate over mini-batches from train_loader
- Forward pass ? compute predictions
- Compute MSE loss
- Backward pass ? compute gradients
- Update weights with Adam optimizer
- Accumulate batch loss for epoch-level tracking
- Validation phase (model.eval() + torch.no_grad()):
- Iterate over test_loader without gradient computation
- Compute MSE loss on test data
- Model checkpointing:
- If validation loss improves, save model weights to best_model.pt

5.3 Training Output

Each epoch prints:

```
epoch $1/100 ==> train loss = X.XXX & valid = X.XXX
```

6. Evaluation

6.1 Best Model Loading

```
model.load_state_dict(torch.load("best_model.pt"))
```

The model with the lowest validation loss across all epochs is loaded for final evaluation.

6.2 Metrics

Metric	Description	Formula

MSE (Mean Squared Error)	Average squared difference between predicted and actual values	$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
R² Score	Proportion of variance in the target explainec	$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \b

6.3 Evaluation Code

```
model.eval()
with torch.no_grad():
    train_pred = model(X_train_tensor)
    test_pred = model(X_test_tensor)
    train_mse = criterion(train_pred, y_train_tensor)
    test_mse = criterion(test_pred, y_test_tensor)

r2 = r2_score(y_test, test_pred)
```

7. Visualization

7.1 Loss Curves

A line plot comparing training loss and validation loss across all 100 epochs is generated using Matplotlib. This helps diagnose:

- Overfitting ? validation loss diverges from training loss
- Underfitting ? both losses remain high
- Good fit ? both losses converge and stabilize

7.2 Predictions vs Actuals

A side-by-side DataFrame of predicted and actual energy values is created for qualitative inspection:

```
pd.concat([predicted_df, actual_df], axis=1)
```

8. Project File Structure

```
powerplant/
?
??? ANN_Regression.ipynb    # Main Jupyter Notebook (full pipeline)
??? powerplant_data.csv     # Dataset (9,568 records, 5 columns)
??? best_model.pt           # Saved PyTorch model weights (best validation loss)
??? README.md               # Project overview
??? PROJECT_DOCUMENTATION.md # This file ? detailed documentation
```

9. How to Run

- Clone/download the project folder.
- Install dependencies:

```
pip install torch pandas numpy scikit-learn matplotlib
```

- Open ANN_Regression.ipynb in Jupyter Notebook or VS Code.
- Run all cells sequentially. The notebook will:
- Load and preprocess data
- Define and train the ANN for 100 epochs
- Save the best model to best_model.pt
- Display loss curves and evaluation metrics

10. Key Concepts Used

Concept	Description
ANN (Artificial Neural Network)	A feed-forward neural network composed of interconnected layers of neur
Regression	Predicting a continuous numerical value (energy output in MW)
Backpropagation	Algorithm to compute gradients of the loss w.r.t. model weights
Adam Optimizer	Adaptive learning rate optimizer combining momentum and RMSProp
MSE Loss	Penalizes large prediction errors quadratically
StandardScaler	Normalizes features to zero mean and unit variance
Mini-batch Gradient Descent	Updates weights using small subsets (batches) of data

Model Checkpointing	Saving the best model during training to prevent losing optimal weights
Train/Validation Split	Separating data to detect overfitting and evaluate generalization

11. Potential Improvements

- Add dropout layers to reduce overfitting
- Experiment with deeper/wider architectures
- Use learning rate scheduling (e.g., ReduceLROnPlateau)
- Apply cross-validation for more robust evaluation
- Add early stopping to avoid unnecessary training epochs
- Hyperparameter tuning with Optuna or GridSearch
- Add RMSE and MAE as additional evaluation metrics