MACHINE LEARNING LECTURE ON ARTIFICIAL NEURAL NETWORK(ANN)

GITHUB LINK: https://github.com/etukuri6/mlcode.git

Student Id:23017408

Name:Naveen

Introduction:

Artificial Neural Networks, or ANNs, are powerful mechanisms for modeling complex relationships among data. By emulating the human brain's overall architecture, they process inputs through a set of interconnected nodes-known as neurons-to find relationships and make predictions. ANNs have been applied to varied tasks: classification, regression, and generative modeling.

This tutorial walks through an ANN for a regression problem with the Boston Housing dataset that predicts the median housing price from the characteristics of a given neighborhood. We'll discuss both the theoretical basis and how to do it practically, along with performance evaluation.

BASIC CONCEPTS OF ANN

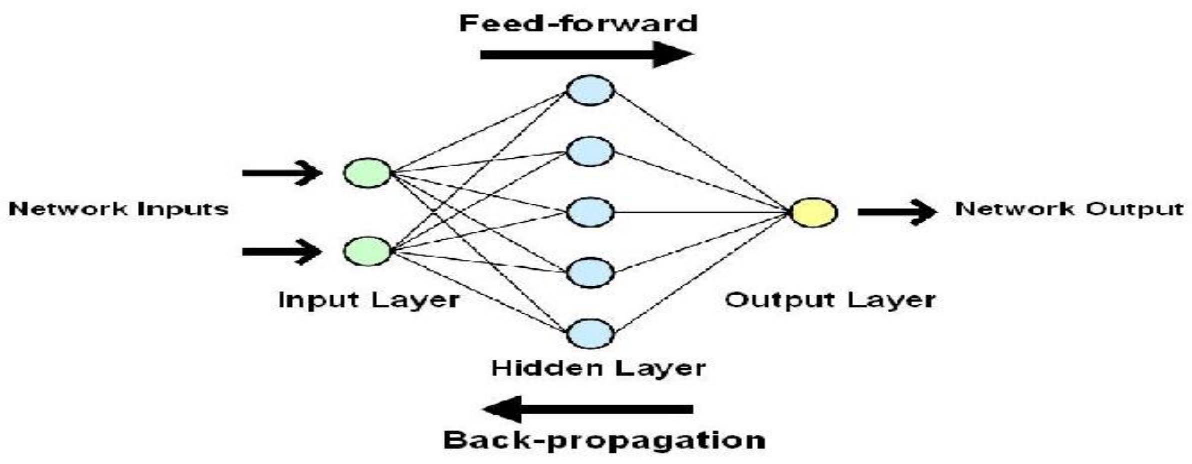


Fig.Image from Internet

1. Structure

An Artificial Neural Network's (ANN) architecture specifies the arrangement of its layers and the interactions between its neurones. ANNs are usually made up of three different kinds of layers:

Input Layer: The dataset provides the first layer with raw input data. To ensure that all data properties are processed, each input layer neurone corresponds to a single feature. Features like the average number of rooms (RM) and the crime rate (CRIM), for example, are fed through the input layer of the Boston Housing dataset.

Hidden Layers: These layers identify patterns and connections in the data to do the heavy work. Each hidden layer neurone applies weights and biases to inputs from the preceding layer before passing the outcome through an activation function. The network can learn non-linear and hierarchical patterns thanks to its many hidden layers. For example, basic associations may be captured by the first hidden layer, and these properties may be further refined into more intricate representations by later layers.

Output Layer: The model's forecast is provided by the last layer. One neurone with a linear activation function is frequently found in the output layer of regression tasks, such as estimating home values.

An ANN's performance is greatly influenced by its architecture, including its number of layers, neurones per layer, and connections.

2. Propagation Forward

The method by which input data moves through the network to produce predictions is called forward propagation. The following actions are involved:

Weighted Sum: Every neurone adds up all of its inputs.

$$z = \sum_{i=1}^{n} \boldsymbol{wixi + b}$$

wi: weights

xi: input values

b: bias

Activation Function: To add non-linearity, the neurone uses an activation function.

$$a=f(z)$$

common activation functions are:

Relu: max (0, z) used in hidden layers.

Sigmoid: utilised for binary classification probability.
Linear: Used for regression tasks in the output layer.
Propagation across Layers: One layer's output becomes the subsequent layer's input. Until the data reaches the output layer, where a final prediction is generated, this process keeps on.

For instance, forward propagation may calculate a weighted sum of features such as RM and LSTAT, apply an activation function, and then output the projected price in a house price prediction model.

3. Optimisation and Backpropagation
In order to reduce the discrepancy between the expected and actual results, backpropagation involves adjusting weights and biases. The following actions are involved:

Loss Function: Indicates the deviation between the actual value and the expected output. Typical loss functions consist of:

MSE, or mean squared error, is used

$$MSE = \sum_{i=1}^{n}(y_i - \widehat{y}_i)^2$$

For classification problems, cross-entropy loss is employed.

Gradient Calculation: The gradients of the loss function with respect to each weight and bias are calculated using the chain rule of calculus. The magnitude and direction of the weight modifications required to lower the inaccuracy are indicated by these gradients.

Updates on Weight: Gradient Descent is one optimisation approach that modifies the weights and biases:

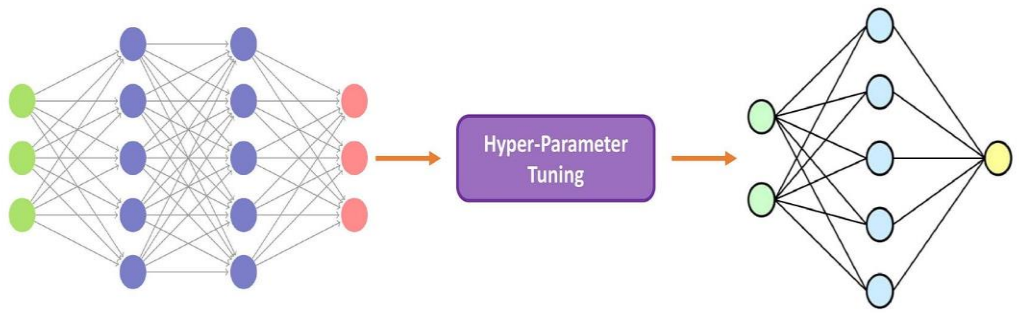
$$w=w-\eta\cdot\partial w/\partial L$$

η: Learning rate

Optimisation Techniques:

SGD, or stochastic gradient descent, uses one data point at a time to update weights.
Adam: For quicker convergence, he combines adaptive learning rates with momentum.
By iteratively fine-tuning the network's parameters, backpropagation allows it to learn from data and generate precise predictions.

Advanced Topic: Hyperparameter Tuning in Artificial Neural Networks



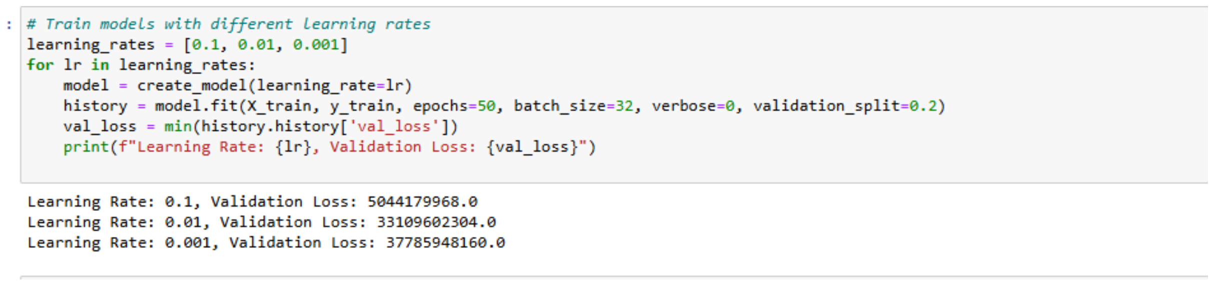
The process of choosing the ideal set of hyperparameters to maximise the performance of a machine learning model is known as hyperparameter tuning. Hyperparameters are predetermined and control the learning process, in contrast to model parameters, which are learnt during training (such as weights and biases in an ANN). Since these hyperparameters have a direct impact on the model's convergence, generalisation, and performance, optimising them is frequently essential to an ANN's success.

This section examines ANN hyperparameters, their significance, and optimisation techniques. Techniques including grid search, random search, and sophisticated methods like Bayesian optimisation are highlighted. To illustrate how hyperparameter adjustment affects a regression problem, the Boston Housing dataset will be used as a case study.

Key Hyperparameters in ANNs

1. Learning Rate

Variation in learning rate is another important hyperparameter in ANNs' training. It defines how large a step the system takes in updating weights according to optimization. A good estimate will result in efficient convergence in training loss towards its minimum while a poor choice may limit performance.



```
# Train models with different learning rates
learning_rates = [0.1, 0.01, 0.001]
for lr in learning_rates:
    model = create_model(learning_rate=lr)
    history = model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)
    val_loss = min(history.history['val_loss'])
    print(f"Learning Rate: {lr}, Validation Loss: {val_loss}")
```

```
Learning Rate: 0.1, Validation Loss: 5044179968.0
Learning Rate: 0.01, Validation Loss: 33109602304.0
Learning Rate: 0.001, Validation Loss: 37785948160.0
```

Fig.from coding part

Small Learning Rates: These make the weight update gradual and stable, allowing convergence after many iterations. However, they increase the training time significantly and also run a great risk of getting stuck in local minima.

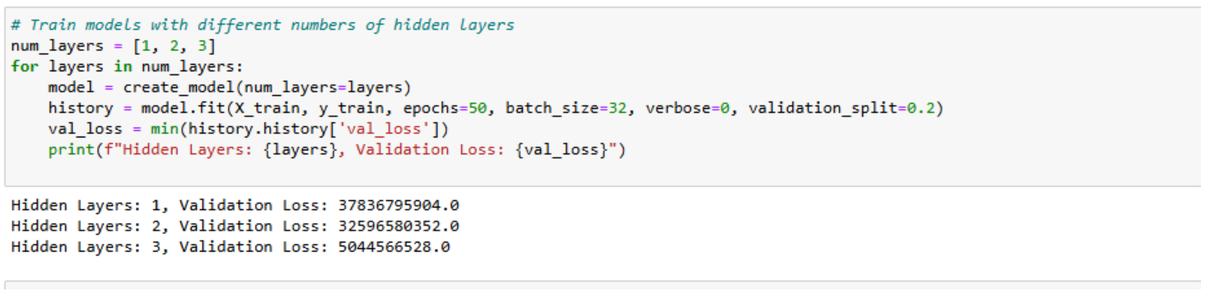
Large Learning Rates: These will speed up the training but may cause overshooting of the model past the optimal point or never converge by continuous oscillations.

Adaptive Learning Rates: Modern optimizers like Adam change the learning rate during training, hence dynamically balancing the speed and stability.

Example Impact: On the Boston Housing dataset, a learning rate of 0.01 converged faster and with a smaller validation loss compared to 0.001, hence the importance of tuning this hyperparameter.

2. Number of Hidden Layers

The number of hidden layers defines the depth of the neural network, hence its capability to model complex patterns. A single-layer network can only represent linear relationships, while deeper networks can capture hierarchical and non-linear relationships.



```
# Train models with different numbers of hidden layers
num_layers = [1, 2, 3]
for layers in num_layers:
    model = create_model(num_layers=layers)
    history = model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)
    val_loss = min(history.history['val_loss'])
    print(f"Hidden Layers: {layers}, Validation Loss: {val_loss}")
```

```
Hidden Layers: 1, Validation Loss: 37836795904.0
Hidden Layers: 2, Validation Loss: 32596580352.0
Hidden Layers: 3, Validation Loss: 5044566528.0
```

Fig.from coding part

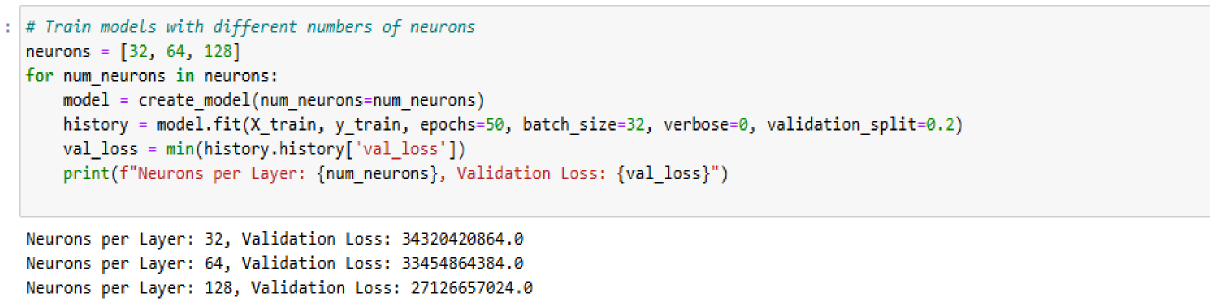
Shallow Networks: These are applicable for simple tasks but they have a tendency to underfit when working on complex datasets such as image or speech recognition.

Deep Networks: These allow for a hierarchical feature extraction but demand more data and computational resources. They are prone to overfitting, which can be mitigated using regularization techniques like dropout.

Optimal Choice: For structured data, 1–3 hidden layers are usually sufficient. On the Boston Housing dataset, adding a second hidden layer improved performance by capturing interactions between neighborhood features like crime rate and average number of rooms.

3. Number of Neurons per Layer

The number of neurons controls the model's capacity to learn patterns. Too few neurons restrict the network's ability to capture complicated relationships, which leads to underfitting. On the other hand, too many neurons increase the risk of overfitting and also the computational cost.



```
# Train models with different numbers of neurons
neurons = [32, 64, 128]
for num_neurons in neurons:
    model = create_model(num_neurons=num_neurons)
    history = model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)
    val_loss = min(history.history['val_loss'])
    print(f"Neurons per Layer: {num_neurons}, Validation Loss: {val_loss}")
```

```
Neurons per Layer: 32, Validation Loss: 34320420864.0
Neurons per Layer: 64, Validation Loss: 33454864384.0
Neurons per Layer: 128, Validation Loss: 27126657024.0
```

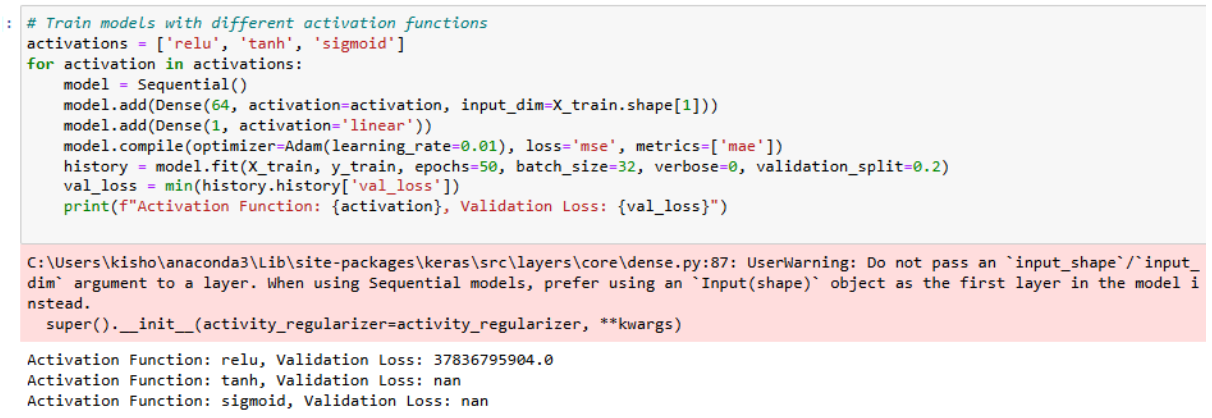
Fig.from coding part

Balancing Act: Start with a reasonable number of neurons, say 64, and adjust based on validation performance. Sparse datasets may call for fewer neurons, while high-dimensional data may require more.

Layer-specific Configuration: Reducing neurons across deeper layers, for instance, 128 → 64 → 32, helps concentrate on successively finer feature representations.

4. Activation Functions

The introduction of non-linearity into the model by activation functions enables it to learn the complex mappings between input and output. Without activation functions, even if the network is deep, it will behave like a linear regression model.



```
# Train models with different activation functions
activations = ['relu', 'tanh', 'sigmoid']
for activation in activations:
    model = Sequential()
    model.add(Dense(64, activation=activation, input_dim=X_train.shape[1]))
    model.add(Dense(1, activation='linear'))
    model.compile(optimizer=Adam(learning_rate=0.01), loss='mse', metrics=['mae'])
    history = model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)
    val_loss = min(history.history['val_loss'])
    print(f"Activation Function: {activation}, Validation Loss: {val_loss}")
```

```
C:\Users\kisho\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_
dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model i
nstead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Activation Function: relu, Validation Loss: 37836795904.0
Activation Function: tanh, Validation Loss: nan
Activation Function: sigmoid, Validation Loss: nan
```

Fig.from coding part

ReLU (Rectified Linear Unit): Efficient and widely used in hidden layers. It avoids vanishing gradients but may suffer from "dead neurons" (output always zero).

Sigmoid: Outputs values between 0 and 1, suitable for binary classification but prone to vanishing gradients.

Softmax: Transforms outputs into probabilities for multi-class classification tasks.

TanH: It maps the values to [-1, 1]. It maintains the gradients better compared to sigmoid.

Application: For regression problems like the Boston Housing dataset, ReLU was used in the hidden layers to learn from complex relationships, while the output layer had a linear activation to predict continuous house prices.

5. Batch Size

Batch size is the number of samples processed before updating model weights. It affects training dynamics, memory usage, and convergence speed.

Small Batch Sizes (e.g., 16, 32): These result in more frequent updates that, in turn, improve generalization. However, they require more iterations and can be computationally expensive.

Large Batch Sizes (e.g., 128, 256): These are computationally efficient but may converge to suboptimal minima, resulting in poorer generalization.

Trade-offs: The choice of the right batch size depends on the size of the dataset and hardware. Example: On the Boston Housing dataset, increasing the batch size from 32 to 64 reduced training time with comparable validation performance.

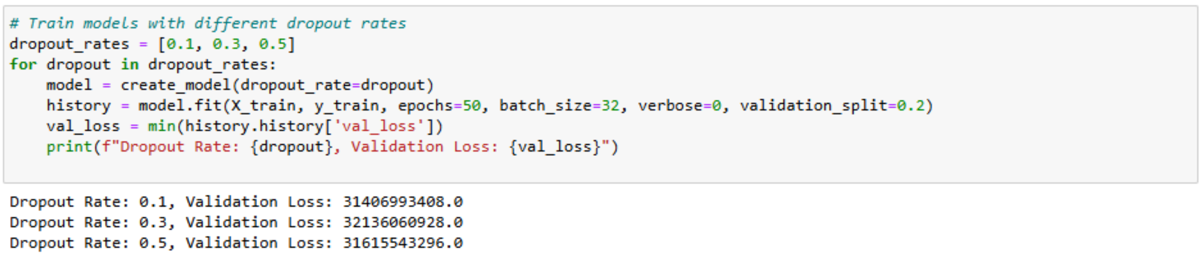
6. Number of Epochs

Epochs specify how many times the model views the entire training dataset. While more epochs allow the model to learn better, they can also result in overfitting, when the return on investment begins to decay. This happens when a model has too little capacity to capture underfitting, or if it has too much and learns noise, resulting in overfitting- when the training error is high, or when it learns noise, leading to high validation error.

Solution: Use early stopping to monitor validation performance and halt training when improvements plateau.

7. Dropout

Dropout is a regularization technique that prevents overfitting by randomly deactivating a fraction of neurons during training. This forces the network to generalize better by not relying too heavily on specific neurons.



```
# Train models with different dropout rates
dropout_rates = [0.1, 0.3, 0.5]
for dropout in dropout_rates:
    model = create_model(dropout_rate=dropout)
    history = model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)
    val_loss = min(history.history['val_loss'])
    print(f"Dropout Rate: {dropout}, Validation Loss: {val_loss}")
```

```
Dropout Rate: 0.1, Validation Loss: 31406993408.0
Dropout Rate: 0.3, Validation Loss: 32136060928.0
Dropout Rate: 0.5, Validation Loss: 31615543296.0
```

Fig.from coding part

Dropout Rate: Typically set between 0.1 and 0.5. Higher rates may impair learning, while lower rates may not provide enough regularization.

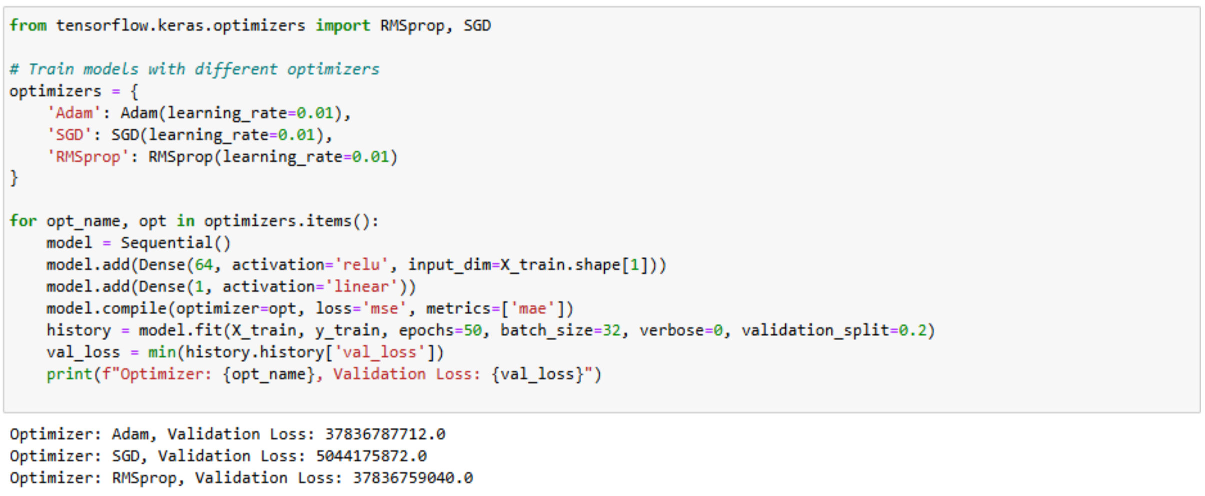
Impact: Dropout works best in large networks where overfitting is more likely.

Example: Applying a 10% dropout rate to the first hidden layer on the Boston Housing dataset improved generalization, reducing the test MAE.

8. Optimizers

Optimizers adjust weights during backpropagation to minimize the loss function. They determine how efficiently the model learns.

SGD (Stochastic Gradient Descent): Simple and effective but may converge slowly.



```
from tensorflow.keras.optimizers import RMSprop, SGD

# Train models with different optimizers
optimizers = {
    'Adam': Adam(learning_rate=0.01),
    'SGD': SGD(learning_rate=0.01),
    'RMSprop': RMSprop(learning_rate=0.01)
}

for opt_name, opt in optimizers.items():
    model = Sequential()
    model.add(Dense(64, activation='relu', input_dim=X_train.shape[1]))
    model.add(Dense(1, activation='linear'))
    model.compile(optimizer=opt, loss='mse', metrics=['mae'])
    history = model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0, validation_split=0.2)
    val_loss = min(history.history['val_loss'])
    print(f"Optimizer: {opt_name}, Validation Loss: {val_loss}")
```

```
Optimizer: Adam, Validation Loss: 37836787712.0
Optimizer: SGD, Validation Loss: 5044175872.0
Optimizer: RMSprop, Validation Loss: 37836759040.0
```

Adam (Adaptive Moment Estimation): Combines momentum and adaptive learning rates for faster and more stable convergence.

RMSprop: Ideal for recurrent neural networks and tasks with sparse gradients.

Application: Adam optimizer was chosen for the Boston Housing dataset due to its ability to handle noisy gradients and dynamic learning rates.

9. Regularization Techniques

Regularization prevents overfitting by constraining model complexity:

L1 Regularization: Adds a penalty proportional to the absolute value of weights, encouraging sparsity.

L2 Regularization (Ridge): Adds a penalty proportional to the square of weights, discouraging large coefficients.

Dropout: Discussed earlier.

Impact: With the combination of dropout, along with L2 regularization on the Boston Housing dataset, it resulted in a more generalized model and reduced overfitting.

10. Advanced Tuning Strategies

Grid Search

Grid search exhaustively tests all combinations of predefined hyperparameter values, ensuring the best configuration is found. However, it is computationally expensive for large search spaces.

Random Search

Random search samples hyperparameters from a distribution and offers faster tuning with comparable results to grid search.

Bayesian Optimization

This advanced method models the objective function and intelligently selects the hyperparameters, while most tools usually require less iterations. This uses tools such as Optuna.

References:

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville's book "Deep Learning"
  A seminal work that covers deep learning theory and applications, including optimisation methods.

- Aurelien Geron’s book "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" Examples of hyperparameter tuning provide useful insights into deep learning and machine learning.

- The documentation for Scikit-Learn
  The official source for hyperparameter tweaking tools such as GridSearchCV and RandomizedSearchCV.

- TensorFlow Documentation for Scikit-Learn
  instructions for using TensorFlow and Keras to develop and fine-tune models.

- Documentation for TensorFlow
  Documentation for Optuna
  An extensive resource for Bayesian optimisation with Optuna.
  Optuna Documents: Instruction and Manuals
  "Practical Neural Network Hyperparameter Tuning with Grid Search and Random Search" provides examples to illustrate how to adjust neural networks.
  Article on TensorFlow Keras Tuner: A Step Towards Data Science
  An instruction manual for Keras Tuner's automated hyperparameter tweaking.
- Datasets for the TensorFlow Guide
  Dataset on Boston Housing
  Carnegie Mellon University hosted the original source:
  Dataset on Boston Housing
  Due to ethical problems, alternative housing databases are

suggested:
Scikit-Learn's California Housing Dataset and OpenML's Ames Housing Dataset Discuss the Boston Housing Dataset's Ethical Issues"Racist Data Destruction?"

- James Bergstra and Yoshua Bengio's research paper "Random Search for Hyper-Parameter Optimisation"draws attention to how much more effective Random Search is than Grid Search.
  Jasper Snoek and colleagues' paper "Practical Bayesian Optimisation of Machine Learning Algorithms" in PDF format.

  A seminal work on Bayesian optimisation methods.
  Courses in Paper PDF

- Andrew Ng's Specialisation in Deep Learning
  discusses hyperparameter tuning, optimisation, and neural networks.
  Fast.ai is the Coursera Link. Coders' Guide to Useful Deep Learning