**Experiment No-6:**

**AIM:**

To perform regularization on a dataset with different hyperparameter values and identifying the best model

**DESCRIPTION:**

Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise, leading to poor performance on unseen data. Regularization is crucial for enhancing the generalization capabilities of machine learning models, allowing them to perform well on new, unseen datasets.

A logo with text and symbols

Description automatically generated

**What is Regularization?**

Regularization introduces a penalty term to the loss function used during model training. This penalty discourages overly complex models by constraining the model's parameters, thus controlling their ability to fit the training data. The primary goal is to improve the model’s ability to generalize beyond the training set, enhancing its performance on unseen data.

Common regularization techniques in deep learning include:

1. **L1 Regularization (Lasso Regression)**:
   * This technique adds a penalty equivalent to the absolute value of the magnitude of coefficients to the loss function. The formulation of the regularized cost function becomes: Cost function=Loss+λ∑∣wi​∣
   * L1 regularization tends to produce sparse weight matrices, meaning it can effectively reduce some weights to zero. This property is beneficial for feature selection, as it can eliminate unnecessary features from the model.
2. **L2 Regularization (Ridge Regression)**:
   * L2 regularization adds a penalty equal to the square of the magnitude of coefficients to the loss function: Cost function= Cost function=Loss+λ∑wi2​
   * This technique discourages large weights but does not necessarily drive them to zero. It is often preferred over L1 regularization because it generally leads to better generalization.
3. **Dropout**:
   * Dropout is a regularization technique that randomly sets a fraction of input units to zero during training. This prevents neurons from co-adapting too much. It can be applied after layers during training and helps to create more robust features by encouraging redundancy in the network.
4. **Early Stopping**:
   * This method involves monitoring the model’s performance on a validation set during training and stopping the training process once the performance ceases to improve. Early stopping helps to avoid overfitting by preventing the model from training too long.

**Implementation Plan**

1. **Dataset Preparation**: We will use a well-known dataset, such as MNIST or CIFAR-10, to facilitate training and evaluation.
2. **Model Design**: A neural network model will be designed with the flexibility to incorporate different regularization techniques. Layers will include activation functions, dropout, and the regularization parameters.
3. **Training the Model**: The models will be trained with varying hyperparameters for the regularization methods. For example:
   * For L1 and L2 regularization, we will adjust the λ values (e.g., 0.01, 0.001, 0.0001).
   * For dropout, we will try different rates.
4. **Evaluate Model Performance**: After training, we will evaluate each model on the validation set, recording accuracy and loss.
5. **Results Analysis**: Finally, we will analyze the results to determine which regularization technique and hyperparameter combination yielded the best performance, highlighting the importance of regularization in developing robust machine learning models.

**CODE:**

import numpy as np

import pandas as pd

# from scipy.misc import imread

A logo with text and symbols

Description automatically generatedimport imageio

from sklearn.metrics import accuracy\_score

from keras.models import Sequential

from keras.layers import Dense, Dropout

from keras.optimizers import Adam, SGD, RMSprop

from keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

# Load and preprocess the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train = x\_train.reshape(-1, 784).astype('float32') / 255.0

x\_test = x\_test.reshape(-1, 784).astype('float32') / 255.0

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Define hyperparameter configurations to test

hyperparams\_list = [

    {'epochs': 10, 'batch\_size': 128, 'optimizer': 'adam', 'dropout': 0.2, 'hidden\_layers': 3},

    {'epochs': 20, 'batch\_size': 64, 'optimizer': 'adam', 'dropout': 0.3, 'hidden\_layers': 4},

    {'epochs': 15, 'batch\_size': 256, 'optimizer': 'rmsprop', 'dropout': 0.4, 'hidden\_layers': 5},

    {'epochs': 12, 'batch\_size': 128, 'optimizer': 'sgd', 'dropout': 0.3, 'hidden\_layers': 4},

    {'epochs': 10, 'batch\_size': 128, 'optimizer': 'adam', 'dropout': 0.5, 'hidden\_layers': 5},

    # Add more configurations as needed

]

# Function to build the model with given hyperparameters

def build\_model(input\_dim, hidden\_layers, dropout\_rate, optimizer\_name):

    model = Sequential()

    for \_ in range(hidden\_layers):

        model.add(Dense(500, activation='relu', input\_dim=input\_dim))

        model.add(Dropout(dropout\_rate))

    model.add(Dense(10, activation='softmax'))  # Output layer

    optimizer = {'adam': Adam(), 'sgd': SGD(), 'rmsprop': RMSprop()}[optimizer\_name]

    model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])

    return model

# Train and evaluate models with each hyperparameter configuration

results = []

for params in hyperparams\_list:

    print(f"Training with params: {params}")

    model = build\_model(784, params['hidden\_layers'], params['dropout'], params['optimizer'])

    history = model.fit(x\_train, y\_train,

                        epochs=params['epochs'],

                        batch\_size=params['batch\_size'],

                        validation\_data=(x\_test, y\_test),

                        verbose=0)

    # Evaluate the model

    score = model.evaluate(x\_test, y\_test, verbose=0)

    print(f"Test accuracy for params {params}: {score[1]:.4f}")

    results.append({

        'epochs': params['epochs'],

        'batch\_size': params['batch\_size'],

A logo with text and symbols

Description automatically generated        'optimizer': params['optimizer'],

        'dropout': params['dropout'],

        'hidden\_layers': params['hidden\_layers'],

        'test\_accuracy': score[1]

    })

# Convert results to DataFrame and display

results\_df = pd.DataFrame(results)

print("\nResults summary:")

print(results\_df)

OUTPUT:

Training with params: {'epochs': 10, 'batch\_size': 128, 'optimizer': 'adam', 'dropout': 0.2, 'hidden\_layers': 3}

Test accuracy for params {'epochs': 10, 'batch\_size': 128, 'optimizer': 'adam', 'dropout': 0.2, 'hidden\_layers': 3}: 0.9788

Training with params: {'epochs': 20, 'batch\_size': 64, 'optimizer': 'adam', 'dropout': 0.3, 'hidden\_layers': 4}

Test accuracy for params {'epochs': 20, 'batch\_size': 64, 'optimizer': 'adam', 'dropout': 0.3, 'hidden\_layers': 4}: 0.9839

Training with params: {'epochs': 15, 'batch\_size': 256, 'optimizer': 'rmsprop', 'dropout': 0.4, 'hidden\_layers': 5}

Test accuracy for params {'epochs': 15, 'batch\_size': 256, 'optimizer': 'rmsprop', 'dropout': 0.4, 'hidden\_layers': 5}: 0.9823

Training with params: {'epochs': 12, 'batch\_size': 128, 'optimizer': 'sgd', 'dropout': 0.3, 'hidden\_layers': 4}

Test accuracy for params {'epochs': 12, 'batch\_size': 128, 'optimizer': 'sgd', 'dropout': 0.3, 'hidden\_layers': 4}: 0.9522

Training with params: {'epochs': 10, 'batch\_size': 128, 'optimizer': 'adam', 'dropout': 0.5, 'hidden\_layers': 5}

Test accuracy for params {'epochs': 10, 'batch\_size': 128, 'optimizer': 'adam', 'dropout': 0.5, 'hidden\_layers': 5}: 0.9773

A screenshot of a computer

Description automatically generated